Entwicklung eines parallelen, adaptiven, komponentenbasierten Strömungskerns für hierarchische Gitter auf Basis des Lattice-Boltzmann-Verfahrens

Von der
Fakultät Architektur, Bauingenieurwesen und Umweltwissenschaften
der Technischen Universität Carolo-Wilhelmina
zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)
genehmigte

Dissertation

von
Sören Freudiger
aus Berlin-Steglitz

Eingereicht am	7. Mai 2009
Mündliche Prüfung am	21. August 2009
Berichterstatter	Prof. Dr.-Ing. habil. Manfred Krafczyk
	Prof. Dr. Ulrich Rüde

2009

„Weihnachtszeit ist Rechenzeit“

Manfred Krafczyk

Danksagung

An dieser Stelle bedanke ich mich bei allen Personen, die mich in den letzten sieben Jahren bei meiner Arbeit am iRMB unterstützt haben und ohne die diese Arbeit nicht zustande gekommen wäre.

Besonderer Dank gilt meinem Doktorvater Prof. Dr.-Ing. habil. Manfred Krafczyk, der mir seit meiner Diplomarbeit beratend zur Seite stand und mir die Promotion überhaupt erst ermöglichte. Insbesondere bedanke ich mich dafür, dass er mir stets die Möglichkeit gab Innovatives auszuprobieren und mich darin auch unterstützte. Seine exzellenten Fachkenntnisse sowie seine oftmals kritischen Anregungen machten die vorliegende Arbeit in dieser Form überhaupt erst möglich.

Herrn Prof. Dr. Ulrich Rüde danke ich für seine Bereitschaft, das Zweitgutachten zu erstellen.

Meinen Kollegen und Kolleginnen am iRMB danke ich ebenfalls für die Unterstützung meiner Arbeit. Im Speziellen bedanke ich mich bei Dr. Jonas Tölke und Maik Stiebler, die mir vor allem bei Fragen numerischer Natur weiterhalfen; bei Sebastian Geller, der mir die Softwareentwicklung näher gebracht hat; bei Jannis Linxweiler, der mir in Bezug auf aktuelle Softwarekonzepte eine große Hilfe war; bei Jan Hegewald, der Dank aufschlussreicher Diskussionen in vielen Belangen weiter helfen konnte; und bei Arne Gessner, der mich bei der Verwirklichung des Serviceframeworks unterstützte.

Zudem bedanke ich mich bei den Korrekturlesern Marc Ermer, der alle meine wissenschaftlichen Arbeiten trotz Fachunkenntnis gewissenhaft korrigierte, und Stefan Donath, der mir insbesondere bei den Performancekapiteln mit seinen Fachkenntnissen beratend zur Seite stand.

Meiner Familie, die vermutlich nicht wusste, was sie vor nunmehr fast 35 Jahren erwartete als sie mich auf diese Welt setzten, gilt besonderer Dank. Sie ermöglichte mir eine sehr gute Ausbildung und unterstützte mich bei jeder meiner oft recht unkonventionellen Entscheidungen.

Der größte Dank gilt jedoch meiner Freundin Manuela, die mir stets die Kraft gab die schönen Zeiten zu genießen und die schlechten zu meistern. Vor allem die von ihr organisierte Südamerikareise verschaffte mir zum richtigen Zeitpunkt die Möglichkeit neue Kraft zu sammeln.

Diese Arbeit wurde von der Deutsche Forschungsgemeinschaft (DFG) gefördert.

Zusammenfassung

Die Simulation von Strömungen ist mittlerweile fester Bestandteil in den meisten Ingenieurbereichen. Innovative mathematisch-physikalische Modelle, effizientere numerische Methoden und leistungsfähigere Computersysteme ermöglichen die Berechnung komplexer Probleme, wodurch die Anforderungen an die Strömungslösersoftware steigen. In dieser Arbeit werden zwei verschiedene Softwarekonzepte für einen Strömungslöser auf der Basis des Lattice-Boltzmann-Verfahrens vorgestellt. Der Fokus liegt darauf, aufzuzeigen wie man vielfältigste Aufgaben, die aus Ansprüchen der Strömungssimulation erwachsen, in einem Code effizient und flexibel unterbringen kann.

Das Ausgangsmodell für die beiden Konzepte stellt das Lattice-Boltzmann-Verfahren dar, das in der hier vorgestellten FD-Diskretisierung auf hierarchischen, kartesischen Gittern Anwendung findet und dessen numerische Grundlagen im ersten Teil der Arbeit besprochen werden. Neben den verschiedenen Einphasenmodellen wird ein neuer Ansatz für die Simulation nicht-mischbarer Zweiphasenströmungen präsentiert. Da nicht-uniforme Gitter Voraussetzung für eine effiziente Berechnung von komplexen Strömungsproblemen sind, ist die diesbezügliche Erweiterung der einzelnen Modelle ebenfalls Schwerpunkt der Grundlagenbesprechung.

Im zweiten Teil der Arbeit wird ein allgemeingültiges, knotenbasiertes Softwarekonzept für einen zwei- und dreidimensionalen Lattice-Boltzmann-Löser vorgestellt. Mit der entwickelten, quad- bzw. octreeartigen Datenstruktur sind zusätzlich zu a priori verfeinerten Gittern auch dynamisch adaptive Gitterstrukturen möglich. Neben der Beschreibung des Softwaredesigns werden zur Validierung des Lösers als auch des Verfahrens verschiedene Strömungsbeispiele präsentiert. Für den Mehrphasenlöser erfolgt die Verifizierung u. a. mittels einer Konvergenzstudie für eine Zweiphasenspaltströmung. Für die Anwendbarkeit des Lösers auf verteilt arbeitenden Computerclustern, wird der 2-D-Rechenkern im letzten Kapitel dieses Teils um einen generalisierten, regelbasierten Parallelisierungsansatz erweitert und validiert.

Als zweites Softwarekonzept wird im letzten Teil der Arbeit ein hybrider Blockgitteransatz, bei dem die Berechnungsknoten in Blöcken vorgehalten werden, als Weiterentwicklung des knotenbasierten Ansatzes vorgestellt. Für einen effizienten numerischen Datenaustausch zur Simulationslaufzeit wird das Connector-Transmitter-Konzept eingeführt, das durch Trennung von Datenermittlung und -austausch eine flexible Interblockkommunikation zulässt. In Hinblick auf Computerclustersysteme mit verteiltem Speicher wird zudem ein geeignetes Interprozessframework ermittelt, das einen dynamischen, objektorientierten Informationsaustausch ermöglicht. Mit diesem wird ein serviceorientiertes, komponentenbasiertes Framework für verteilte, adaptive Simulationen auf Basis der 3-D-Blockgitterstruktur für verschiedene Lattice-Boltzmann-Modelle umgesetzt. Hier werden die entwickelten Servicekomponenten sowie die implementierten Algorithmen und Konzepte im Detail vorgestellt. Für den Prototypen werden abschließend die parallele Leistungsfähigkeit ermittelt und aktuelle Anwendungsbeispiele präsentiert.

Abstract

In this work, two different software concepts for efficient and flexible flow simulations based on the Lattice Boltzmann (LB) method are presented.

In the first part of this work, the basics of the LB method and the underlying finite difference discretization on hierarchical, Cartesian grids are described. Apart from single phase flows a new approach for immiscible two phase flows is derived. The second part of this thesis introduces a general, node based software concept for a two dimensional and three dimensional flow solver. The underlying quad- and octree data structure allows a dynamic and adaptive update of the grid structure as an extension to a priori refined grids. The software design also supports massively parallel simulations on distributed systems by a generalized, rule based parallelization approach. Subsequent to the description of concept, validations for the LB method and the flow solver are presented.

In the third part of this thesis, an improved software concept is introduced. The proposed hybrid block grid approach enhances the node based approach. The connector transmitter concept separates the data collection from the data distribution between two blocks and allows an efficient and flexible numerical data transfer during the simulation. After the selection of a dynamic and object oriented interprocess framework, the implementation of a service oriented and component based framework is presented. The framework provides distributed, adaptive simulations on a 3-D block grid structure for various LB models. The developed service components as well as the implemented algorithms and concepts are described in detail. Finally, parallel efficiency is investigated and flow examples are presented.

Abkürzungsverzeichnis

Abkürzungen

2-D	Zweidimensional
3-D	Dreidimensional
AABB	Axis Aligned Bounding Box
Abb.	Abbildung
Anh.	Anhang
Alg.	Algorithmus
API	Application Programming Interface
BMR	Block Mesh Refinement
BC	Boundary Condition
Abs.	Absatz
Abschn.	Abschnitt
bzw.	beziehungsweise
CFD	Computational Fluid Dynamics
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DFG	Deutsche Forschungsgemeinschaft
d. h.	das heißt
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
engl.	englisch
etc.	et cetera
FSI	Fluid-Struktur-Interaktion
GCC	GNU Compiler Collection
ggf.	gegebenenfalls
Gl.	Gleichung
FD	Finite Differenzen
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
Ice	Internet Communications Engine
i. d. R.	in der Regel
IDL	Interface Definition Language
IIOP	Internet Inter ORB Protocol
IOR	Interoperable Object Reference
IPC	Interprocesscommunication
Kap.	Kapitel
Korr.	Korrektur

RK	Rothmann-Keller
LB	Lattice-Boltzmann
LES	Large-Eddy-Simulation
MPI	Message Passing Interface
MRT	Multi Relaxation Time
NS	Navier-Stokes
nups	nodal updates per second
OMG	Object Management Group
OO	Objektorientierung
ORB	Object Request Broker
POA	Portable Object Adapter
RPC	Remote Procedure Call
SirAnn	Sören
Slice	Specification Language für Ice
SOA	serviceorientierte Architektur
SRT	Single Relaxation Time
St	Strouhal-Zahl
STL	Standard Template Library
Tab.	Tabelle
u. a.	unter anderem
UDDI	Universal Description, Discovery and Integration
u. U.	unter Umständen
vgl.	vergleiche
VOF	Volume of Fluid
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language Remote Procedure Call
z. B.	zum Beispiel

Superskripte

$corr$	Korrektur
eq	Gleichgewicht
neq	Nichtgleichgewicht
OS	Oberflächenspannung
T	Transponent
Zyl	Zylinder

Indizes

$analyt$	analytisch
b	blau
BC	Boundary Condition

f	fein
g	grob
i	diskrete Richtung
inv	entgegengesetzte Richtung
$\mathbf{i}_L$	levelbezogene Indizes
l	Level
m	gemittelt
max	maximal
min	minimal
N	Nenner
num	numerisch
r	rot
w	Wand
Z	Zähler

Symbole

$\mathbf{C}$	Phasenfeldgradient
$c_{D,L}$	Kraftbeiwerte (*drag, lift*)
c	Propagationsgeschwindigkeit
c_s	Schallgeschwindigkeit
Δ	Wertdifferenz
$\mathbf{e}$	diskrete Geschwindigkeiten
e	Entmischungsfaktor
Eo	Eötvös-Zahl
ϵ	Dissipation
η	Normale
F	Kraft
f	Teilchenaufenthaltswahrscheinlichkeit (Verteilung)
f^{eq}	Gleichgewichtsverteilung
f_{inv}	entgegengerichtete Verteilung von f_i
$\tilde{f}$	Postkollisionsverteilung
g	vereinfachte Boltzmanngleichng
Γ	Rand
γ	Viskositätsverhältnis
h	Höhe
j	Impuls
l	Level
M	Transformationsmatrix
$\mathbf{m}$	Momentenvektor
$\mathbf{m}^{eq}$	Gleichgewichtsmomentenvektor
M	Morton-Zahl
$n_{l,\alpha}$	normierter Phasenfeldgradient
#	Anzahl an/von
μ	dynamische Viskosität

ν	kinematische Viskosität
Ω	Kollisionsoperator
ω	Kollisionsfaktor
p	Druck
ϕ	Ordnungsparameter
$\mathbf{\Phi}$	Basisverktoren
Ψ	Potentialfunktion
ψ	Phasenfeld
q	Anzahl an Kollokationspunkten
$\mathbf{q}$	normierte Abstände
q_i	normierter Abstand in Richtung i
$q_{x_{1/2/3}}$	Wärmefluss
Re	Reynoldszahl
ρ	makroskopische Dichte
S	Diagonalmatrix mit Relaxationszeiten (Kollisionsmatrix)
$S_{\alpha\beta}$	Spannungstensor
σ	Oberflächenspannung
$s_{i,i}$	Kollisionsparameter
T	Temperatur
τ	Relaxationszeit
t	Zeit
$\mathbf{u}$	makroskopische Geschwindigkeit
w_i	Wichtungsfaktor
$\mathbf{w}$	Wichtungsfaktoren
$\mathbf{x}$	Ort
$x_{1,2,3}$	Richtungskomponenten
$\boldsymbol{\xi}$	mikroskopische Geschwindigkeit

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In konstruktiven Ingenieurbereichen, wie dem Bau- oder dem Maschinenbauwesen, ist eine genaue Erfassung von Strömungsphänomenen von großer Bedeutung. Mit Hilfe der Aerodynamik als Teil der Fluiddynamik werden beispielsweise bei Planung neuer Hochhäuser Vorhersagen über die Einflüsse von Luftströmungen (Wind) auf Gebäude sowie Aussagen über mögliche Auswirkungen resultierender Strömungsänderungen auf bereits bestehende Gebäude getroffen. Bei der Ermittlung kommen oft Miniaturmodelle zum Einsatz, mit deren Hilfe durch (Windkanal-)Versuche Aussagen bzw. Rückschlüsse auf die Realität getroffen werden. Abgesehen von der Tatsache, dass Modelländerungen meist mit einem hohen zeitlichen wie auch materiellen und somit finanziellen Aufwand einhergehen, ist eine Projektion auf die Realität aufgrund deren Komplexität nicht immer ohne weiteres möglich.

Seit langem hat sich daher die numerische Strömungsmechanik, Computational Fluid Dynamics (CFD), als adäquates Hilfsmittel etabliert. Heutzutage werden Gebäude und Konstruktionen mit der Unterstützung von Computersimulationen geplant. Stahlkonstruktionen werden mittels Strukturlösersoftware hinsichtlich ihrer Stabilität für verschiedenste Last- und Versagensfälle überprüft. Automobilhersteller optimieren mit Hilfe von Strömungssimulationen den Luftwiderstand und den Anpressdruck der Autokarosserien. Die Vorteile einer derartigen virtuellen Abbildung sind unter anderem kostengünstige Entwicklung, größtmögliche Flexibilität, schnelle Verwirklichung von Modelländerungen sowie die Berücksichtigung einer Vielzahl von Parametern.

CFD findet bereits Anfang des letzten Jahrhunderts zum ersten Mal Erwähnung. Industriell kam CFD jedoch erst Mitte der 1980er Jahre im Flugwesen zum Einsatz, in anderen Industriebereichen sogar erst ein gutes Jahrzehnt später [18]. Die meisten Softwarelösungen basieren auf Näherungslösungen der allgemein hin als gültig anerkannten Navier-Stokes-Gleichungen. Die EXA Corporation konnte vor einigen Jahren mit PowerFLOW [36] vor allem in der Automobilindustrie ein Softwareprodukt etablieren, dessen Lösungen auf einem neuen, gänzlich anderen Ansatz beruhen. Die dort verwendete Lattice-Boltzmann-Methode ist vergleichsweise jung und ebenfalls Gegenstand aktueller Forschungen.

Auch wenn bereits viele Strömungsprobleme mittels numerischer Simulation gelöst werden können, sind jedoch noch mindestens genau so viele Fragen offen. Dies gilt insbesondere für turbulente, also zeitlich und räumlich ungeordnet ablaufende Strömungen. Hinzu kommt, dass sowohl die numerische Lösung als auch deren softwaretechnische Umsetzung nicht trivial sind. Oft werden bereits für die Simulation kleiner Probleme so viel Rechenressourcen benötigt, dass sie mit Standardcomputern überhaupt nicht oder nicht effektiv gelöst werden können. Hier können Computercluster, also ein Verbund von in der Regel homogenen Rechnern, die über ein Netzwerk miteinander kommunizieren, Abhilfe schaffen. Dank sinkender Hardwarepreise und Leistungssteigerung bei den Netzwerkarchitekturen stieg deren Verbreitung in den letzten beiden Jahrzehnten sprunghaft an. Zugleich sind jedoch viele Numeriker mit den ebenfalls ansteigenden Anforderungen an die eigentliche Software überfordert. So müssen z. B. nicht-uniforme Berechnungsgitter für komplexe Geometrien einfach und schnell zu generieren sein. Sie müssen sich lokal hinsichtlich der strömungscharakteristischen

Längen mit geringem Aufwand anpassen lassen. Darüber hinaus ist eine Programmparallelisierung heutzutage unabdingbar und die damit verbundene Problematik der Nachbearbeitung der großen Datenmengen ist ebenfalls Gegenstand aktueller Forschung. Hier bedarf es eines Softwareentwicklers, der als Schnittstelle zwischen der Hardware und dem Forscher fungiert und diesem für derartige Systeme eine möglichst allgemeingültige und flexible Lösung zur Verfügung stellt. Die Hauptmotivation dieser Arbeit war es diesbezüglich Lösungsansätze für nicht-triviale Simulationen auf Basis der Lattice-Boltzmann-Methode zu entwickeln. Eine Herausforderung war dabei die Implementierung eines universellen, nicht-uniformen, dynamisch veränderbaren Finite-Differenzen-Netzes für Strömungssimulationen sowohl in 2-D als auch in 3-D. Auf diesen Gittern sollten beliebige, gegebenenfalls mit Randbedingungen behaftete, geometrische Objekte abgebildet werden, die in Hinblick auf eine Fluid-Struktur-Interaktionen (FSI) frei beweglich/verformbar sind. Zudem sollte ein allgemeingültiges, komponentenbasiertes Framework geschaffen werden, das verteilte Simulationen sowie die freie Wahl des Übertragungsstandards für den numerischen Interprozessdatenaustausch ermöglicht und darüber hinaus dynamische Gitteränderungen zur Laufzeit zulässt.

Bisherige Implementierungen stellen in der Regel hinsichtlich Simulationslaufzeit ("Performance") und Plattform optimierte Lösungen dar, die hauptsächlich mit uniformen Gittern arbeiten. Diese sind, wenn überhaupt, nur mit sehr viel Aufwand für andere Lattice-Boltzmann-Modelle (Mehrphasenprobleme, freie Oberfläche, Turbulenz, etc.) zu erweitern. Zudem verwenden diese meist prozedurale Sprachen, wie Fortran oder C. In dieser Arbeit wurde bewusst ein objektorientierter Ansatz gewählt, da durch dessen Modularisierung eine realitätsnahe Modellierung ermöglicht sowie die Wiederverwendbarkeit von Komponenten vereinfacht wird. Dies ist insbesondere hinsichtlich dynamisch veränderlichen Datenstrukturen von Vorteil. Als Programmiersprache wurde C++ gewählt, da sie neben der objektorientierten und prozeduralen Programmierung mit Hilfe der Template-Metaprogrammierung auch das generische Programmierparadigma unterstützt. Dies ermöglicht eine abstrakte und effiziente Programmierung.

1.2 Gliederung der Arbeit

Der erste Teil der Arbeit (Teil I) befasst sich mit den notwendigen numerischen Grundlagen des Lattice-Boltzmann-Verfahrens für Einphasenprobleme. Da der Fokus der Arbeit auf der Entwicklung eines auf hierarchischen Gittern basierenden Lösers liegt, werden hier auch die notwendigen Erweiterungen hinsichtlich der Berücksichtigung von unterschiedlichen Gitterauflösungen in 2-D und 3-D besprochen. Erhöhte Anforderungen an den Löser liegen vor, wenn nicht-lokale, numerische Operationen, wie die Berechnung von Gradienten, durchgeführt werden müssen. Aus diesem Grund wird anschließend die Erweiterung des Berechnungskerns für Mehrphasenprobleme besprochen. Bei der Validierung dieses Modells zeigt sich, dass für das Lösen von Mehrphasenproblemen mit diesem eine adaptive Gitterstruktur von Vorteil ist.

Der zweite Teil der Arbeit (Teil II) beschäftigt sich mit einem rein knotenbasierten Lösungsansatz. Dieser bietet durch eigenständige Knotenobjekte größtmögliche Flexibilität hinsichtlich der Erweiterbarkeit und Anpassungsfähigkeit anderer Kerne. Bewusst wurde hier auf die Trennung von Topologie und Physik geachtet und somit die Möglichkeit geschaffen, den Ansatz für weitere Finite-Differenzen-Löser zu erweitern. Um die Flexibilität für verteilte Berechnungen aufrecht zu erhalten, wurde für den 2-D-Löser ein abstraktes, regelbasiertes Framework für die Parallelisierung von Simulationen mit a priori verfeinerten Gittern entwickelt und validiert. Eine besondere Herausforderung war die adaptive Erweiterung des 3-D-Mehrphasenkerns, bei dem sich das Gitter zur Laufzeit kontinuierlich an das Strömungsproblem anpasst. Auch hier wurde eine allgemeingültige

Lösung angestrebt, wodurch die topologischen und geometrischen Algorithmen allen Kernen zur Verfügung stehen und die Verfeinerungs- bzw. Vergröberungsindikatoren vom Anwender frei gewählt werden können.

Im Laufe der Entwicklung des rein knotenbasierten Ansatzes zeigten sich einige Einschränkungen. So kann eine optimale Rechenleistung nur unter Verwendung optionaler, starrer Datenstrukturen erzielt werden. Diese wirken sich jedoch negativ auf die gewünschte Anpassungsfähigkeit aus. Die beliebigen topologischen Strukturen des Knotengitters haben zudem den Nachteil, dass eine Kombination aus verteilter und adaptiver Simulation eine zu große Komplexität sowohl hinsichtlich Effizienz als auch der softwaretechnischen Umsetzung bedeuten würde. Aus diesem Grund wurde ein hybrider Blockansatz entwickelt, der im dritten Teil der Arbeit (Teil III) vorgestellt wird.

Zunächst wird die in Hinblick auf eine dynamische, objektorientierte Interprozesskommunikation geeignete Middleware-Software ermittelt, die später bei verteilten Anwendungen zum Einsatz kommen soll. Anschließend wird das neue Softwarekonzept vorgestellt, das dem knotenbasierten Ansatz sehr ähnlich ist. Allerdings stellen jetzt nicht mehr Knoten sondern quaderförmige, eine Submenge von Knoten enthaltende Blöcke die kleinste Einheit des Gitters dar. Topologische Algorithmen, wie das Verfeinern und Vergröbern der Blöcke sowie die Gebietszerlegung bei verteilten Simulationen, finden ausschließlich auf diesen Blöcken Anwendung. In diesem Zusammenhang wird das hier entwickelte Connector-Transmitter-Konzept vorgestellt, das den eigentlichen Datenaustausch zwischen den Blöcken vollständig von der Datenermittlung/-verteilung entkoppelt. Bei der Kernentwicklung muss somit nicht mehr zwischen lokaler und prozessübergreifender Interblockkommunikation unterschieden werden. Dies minimiert sowohl die Komplexität des Softwarecodes als auch Quellcoderedundanz.

Im Anschluss werden die sogenannten Interaktormodule beschrieben, mit deren Hilfe Geometrieobjekte mit den jeweiligen Randbedingungen auf das Gitter abgebildet werden. Diese Module dienen drüber hinaus auch zur Gittergenerierung und zur Fluid-Struktur-Interaktion zur Berechnungslaufzeit.

Die letzten Kapitel beschäftigen sich mit der Entwicklung eines serviceorientierten, komponentenbasierten Frameworks für verteilte, adaptive Simulationen auf Basis der 3-D-Blockgitterstruktur für verschiedene Lattice-Boltzmann-Modelle. Hier werden die entwickelten Servicekomponenten sowie die implementierten Algorithmen und Konzepte im Detail vorgestellt. Durch die Verwendung des TCP/IP-Protokolls zur Kommunikation zwischen den Services ist das Framework nahezu überall einsetzbar. Optional kann der numerische Datenaustausch zwischen den einzelnen Recheneinheiten zur Simulationslaufzeit über ein separates, effizienteres Protokoll/Framework erfolgen. Hierzu wurde für den entwickelten Prototypen unter Verwendung verschiedener Interprozessbibliotheken die jeweilige Rechenleistung ermittelt. Die praktische Verwendung des Frameworks wird anhand verschiedener Anwendungsbeispiele vorgestellt.

Die Arbeit schließt mit einer Zusammenfassung und gibt Ausblick auf mögliche Erweiterungen.

Teil I

Die Lattice-Boltzmann-Methode

Überblick

In diesem Teil der Arbeit werden die theoretischen Grundlagen und Begrifflichkeiten der Lattice-Boltzmann-Methode erläutert. Diese stellen die Basis für die Rechenkerne des später vorgestellten Strömungssimulators VIRTUALFLUIDS und dessen Softwarekonzepte dar. Nach einer kurzen Einleitung, die unter anderem den Zusammenhang zwischen der Lattice-Boltzmann- und den Navier-Stokes-Gleichungen erläutert, wird auf die Theorie für Einphasenströmungen eingegangen. Hier wird neben der eigentlichen Vorgehensweise und den in dieser Arbeit verwendeten Randbedingungen auch die Erweiterung hinsichtlich nicht-regulärer Gitter vorgestellt. Erhöhte Anforderungen stellt die Implementierung eines Lösers für Mehrphasenprobleme dar. Hierfür werden aktuelle Ansätze besprochen und anschließend das im Rahmen dieser Arbeit validierte erweiterte Rothmann-Keller-Mehrphasenmodell vorgestellt. Anhand erster Ergebnisse werden die Problematik dieses Modells für nicht-uniforme Gitter und die daraus resultierenden Anforderungen an den Multiphysik-Löser VIRTUALFLUIDS erläutert.

2 Grundlagen der Lattice-Boltzmann-Methode

Zur Beschreibung eines Newton'schen Fluids haben sich seit langem die Navier-Stokes(NS)-Gleichungen bewährt. Diese erfüllen die aus der makroskopischen Betrachtungsweise geforderte Massen-, Impuls- und Energieerhaltung. In den letzten Jahren konnte sich die Lattice-Boltzmann(LB)-Methode neben numerischen Verfahren auf Basis der Navier-Stokes-Gleichungen für die Berechnungen von Strömungen etablieren. Aus historischer Sicht entwickelte sich die LB-Methode aus den zellulären Lattice-Gas-Automaten [104, 165]. Der Zusammenhang zu der 1872 von Ludwig Boltzmann aufgestellten Boltzmann-Gleichung wurde erst später nachgewiesen [69, 143]. Die Boltzmann-Gleichung geht von einer mesoskopischen Betrachtungsweise aus, indem sie die raumzeitliche Entwicklung von Teilchenaufenthaltswahrscheinlichkeiten f beschreibt [16, 17]:

$$\frac{\partial f}{\partial t} + \boldsymbol{\xi} \cdot \frac{\partial f}{\partial \mathbf{x}} + \mathbf{F} \cdot \frac{\partial f}{\partial \boldsymbol{\xi}} = \Omega(f, f') \tag{2.1}$$

Die Größe $f(t, \boldsymbol{\xi}, \mathbf{x})$ gibt die Wahrscheinlichkeit an, ein Teilchen zum Zeitpunkt t mit der Geschwindigkeit $\boldsymbol{\xi}$ am Ort $\mathbf{x}$ anzutreffen. Die ersten beiden Terme der linken Seite modellieren den advektiven Teilchentransport mit der (kontinuierlichen) mikroskopischen Geschwindigkeit $\boldsymbol{\xi}$. Einflüsse durch äußere Kräfte $\mathbf{F}$ werden durch den dritten Term berücksichtigt.

Der sogenannte Kollisionsoperator Ω modelliert die Interaktion der Teilchen. Er macht die Boltzmann-Gleichung zu einer komplexen Integro-Differentialgleichung, da die Teilchenaufenthaltswahrscheinlichkeiten nicht nur vom Ort, sondern auch von der mikroskopischen Geschwindigkeit $\boldsymbol{\xi}$ abhängen. Es lässt sich jedoch zeigen, dass sich die Teilchenaufenthaltswahrscheinlichkeiten bei einem Gas im uniformen Zustand mit Hilfe der Maxwellverteilung beschreiben lassen. Ist man nicht an den Details des Kollisionsprozesses interessiert und ist das Gas nicht weit vom Gleichgewichtszustand entfernt, so kann zum Beispiel der von Bhatnagar, Gross und Krook (BGK) vereinfachte Kollisionsoperator verwendet werden [7, 122]:

$$\Omega = -\frac{1}{\tau}\left(f - f^{eq}\right) \tag{2.2}$$

Eine gebräuchliche Approximation für die Gleichgewichtsverteilung f^{eq} ist die polynomiale Annäherung niederer Ordnung des sich einstellenden Maxwell'schen Gleichgewichts. τ entspricht der Relaxationszeit, also der mittleren Stoßzeit zweier Moleküle bzw. der mittleren Zeit, in der sich ein Molekül ohne Kollision bewegt. Die makroskopischen Größen Dichte ρ und Impuls $\rho\mathbf{u}$ ergeben sich als Momente der Verteilungsfunktion bezüglich der mikroskopischen Geschwindigkeit $\boldsymbol{\xi}$:

$$\rho(t, \mathbf{x}) = \int_{-\infty}^{\infty} f(t, \boldsymbol{\xi}, \mathbf{x})\, d\boldsymbol{\xi} \tag{2.3}$$

$$\rho(t, \mathbf{x})\, u_\alpha(t, \mathbf{x}) = \int_{-\infty}^{\infty} \xi_\alpha f(t, \boldsymbol{\xi}, \mathbf{x})\, d\boldsymbol{\xi} \tag{2.4}$$

Die *vereinfachte Boltzmann-Gleichung* lautet somit unter Vernachlässigung des Einflusses äußerer Kräfte:

$$\frac{\partial f(t, \boldsymbol{\xi}, \mathbf{x})}{\partial t} + \boldsymbol{\xi}\, \frac{\partial f(t, \boldsymbol{\xi}, \mathbf{x})}{\partial \mathbf{x}} = -\frac{1}{\tau}\Big(f(t, \boldsymbol{\xi}, \mathbf{x}) - f^{eq}\big(\rho(t, \mathbf{x}), \mathbf{u}(t, \mathbf{x})\big)\Big) \tag{2.5}$$

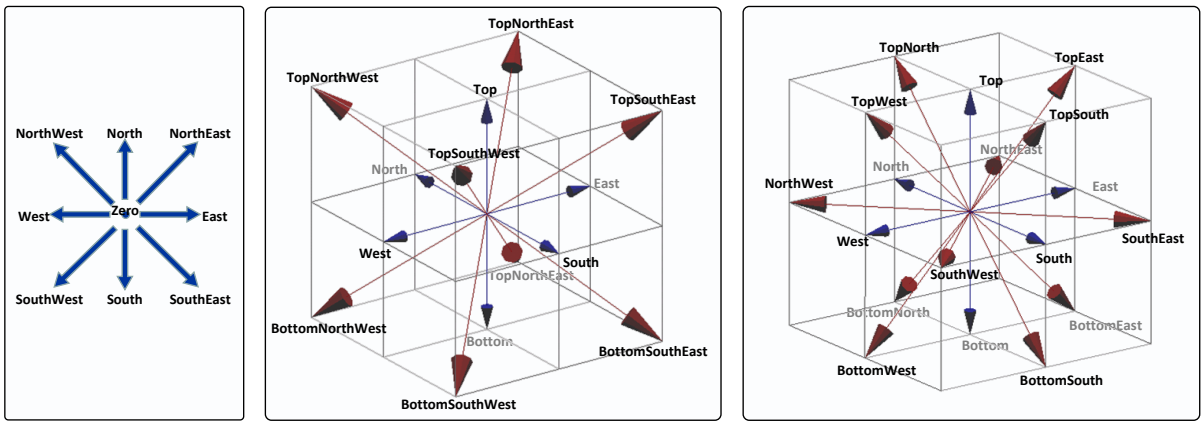
Eine Diskretisierung bezüglich des Geschwindigkeitsraumes führt zur *diskreten Boltzmann-Gleichung* (Discrete Velocity Model) mit den diskreten mikroskopischen Geschwindigkeiten $\mathbf{e}_i$ [69]:

$$\frac{\partial f_i(t,\mathbf{e_i},\mathbf{x})}{\partial t} + \mathbf{e}_i \frac{\partial f_i(t,\mathbf{e}_i,\mathbf{x})}{\partial \mathbf{x}} = -\frac{1}{\tau}\Big(f(t,\mathbf{e}_i,\mathbf{x}) - f^{eq}\big(\rho(t,\mathbf{x}),\mathbf{u}(t,\mathbf{x})\big)\Big) \tag{2.6}$$

Die eigentliche *Lattice-Boltzmann-Gleichung* erhält man nach einer weiteren Diskretisierung in Raum und Zeit z. B. mit Hilfe von Finite-Differenzen:

$$f_i(t+\Delta t,\mathbf{x}+\mathbf{e}_i\Delta t) = f_i(t,\mathbf{x}) - \frac{\Delta t}{\tau}\left(f_i(t,\mathbf{x}) - f_i^{eq}\left(\rho(t,\mathbf{x}),\mathbf{u}(t,\mathbf{x})\right)\right) \quad i = 0,\ldots,(q-1) \tag{2.7}$$

q entspricht der Anzahl der diskreten Geschwindigkeitsvektoren $\mathbf{e}_i$ des verwendeten LB-Modells (Abb. 2.1).



(a) *d2q9* ***(b)*** *d3q15* ***(c)*** *d3q19*

Abbildung 2.1: *Beispiele für LB-Diskretisierungssterne*

Die Modellnotation geht dabei auf Qian zurück, der erstmalig diese Diskretisierung durchgeführt hat [122]. d ist die Dimension des Raumes und q die Anzahl der Kollokationspunkte bzw. die Anzahl der Verteilungen pro Gitterknoten.

Analog zu Gl. 2.3 und 2.4 ergeben sich Masse und Impuls:

$$\rho = \sum_i f_i \tag{2.8}$$

$$\rho\,\mathbf{u} = \sum_i \mathbf{e}_i f_i \tag{2.9}$$

Den Spannungstensor $S_{\alpha\beta}$

$$S_{\alpha\beta} = \nu\rho\left(\frac{\partial u_\alpha}{\partial x_\beta} + \frac{\partial u_\beta}{\partial x_\alpha}\right) \tag{2.10}$$

erhält man mit

$$S_{\alpha\beta} = -\left(1 - \frac{\Delta t}{2\tau}\right)\sum_i e_{i\alpha}\, e_{i\beta}\,\left(f_i - f_i^{eq}\right) \tag{2.11}$$

Für kleine Knudsen- und Machzahlen können, ausgehend von der Boltzmann-Gleichung, mit Hilfe der Chapman-Enskog-Multiskalenentwicklung die Navier-Stokes-Gleichungen und die Kontinuitätsgleichung hergeleitet werden [40, 87, 122]. Somit erhält man unter Berücksichtigung dieser Ein-

schränkungen aus Näherungslösungen der Boltzmann-Gleichung entsprechende Näherungslösungen der Navier-Stokes-Gleichungen (Abb. 2.2). In [87] wird anhand einer asymptotischen und einer Taylor-Entwicklung mit Hilfe der advektiven Skalierung gezeigt, dass es sich bei dem Lattice-Boltzmann-Verfahren um ein Verfahren zweiter Ordnung in Raum und Zeit für die kompressiblen Navier-Stokes-Gleichungen und unter Verwendung diffuser Skalierung, um ein Verfahren erster Ordnung in der Zeit und zweiter Ordnung im Raum bezüglich der inkompressiblen Navier-Stokes-Gleichungen handelt. Ein Vergleich der beiden Verfahren ist in [46] und [102] zu finden.

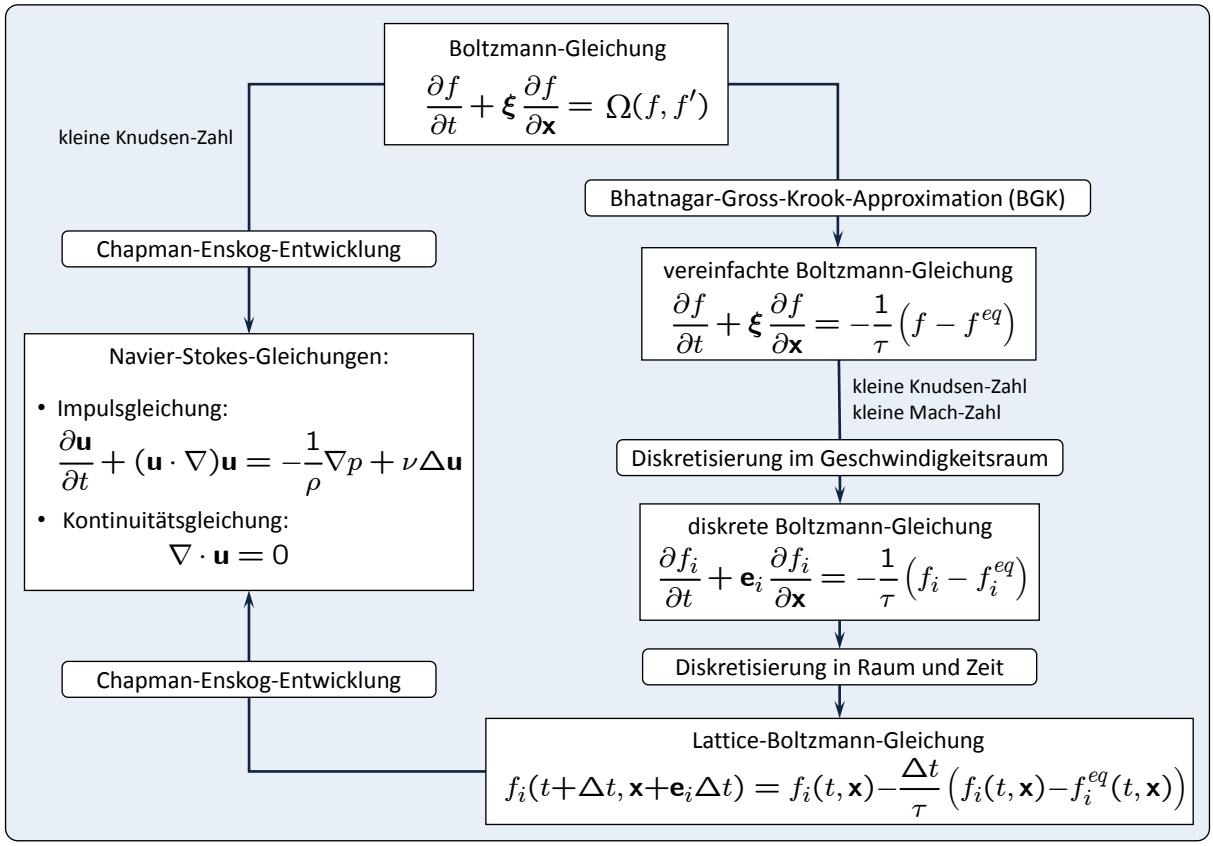


Abbildung 2.2: *Beziehung Boltzmann-, Lattice-Boltzmann- und Navier-Stokes-Gleichung(en)*

3 Lattice-Boltzmann-Methode für Einphasenprobleme

Das explizite Finite-Differenzen-Verfahren beschreibt die Advektion und Kollision von Teilchendichten f_i auf einem regulären Gitter, wobei die diskreten Geschwindigkeitsvektoren $\mathbf{e}_i$ mit einer frei wählbaren Geschwindigkeitskonstante c skaliert sind. Die Raum- und Zeitskalierung sind durch $\Delta x = c\,\Delta t$ gekoppelt.

$$\{\mathbf{e}_{i,d2q9}\} = \begin{Bmatrix} 0 & c & 0 & -c & 0 & c & -c & -c & c \\ 0 & 0 & c & 0 & -c & c & c & -c & -c \end{Bmatrix}$$

$$\{\mathbf{e}_{i,d3q19}\} = \begin{Bmatrix} 0 & c & -c & 0 & 0 & 0 & 0 & c & -c & c & -c & c & -c & c & -c & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & -c & 0 & 0 & c & -c & -c & c & 0 & 0 & 0 & 0 & c & -c & c & -c \\ 0 & 0 & 0 & 0 & 0 & c & -c & 0 & 0 & 0 & 0 & c & -c & -c & c & c & -c & -c & c \end{Bmatrix}$$

c legt zugleich die Größenordnung der Schallgeschwindigkeit c_s fest:

$$c_s = \frac{1}{\sqrt{3}}\,c \tag{3.1}$$

Die Richtungen werden in dieser Arbeit wie folgt indiziert und entsprechend ihrer jeweiligen Himmelsrichtung deklariert:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$d2q9$	Z	E	N	W	S	NE	NW	SW	SE	-	-	-	-	-	-	-	-	-	-
$d3q19$	Z	E	W	N	S	T	B	NE	SW	SE	NW	TE	BW	BE	TW	TN	BS	BN	TS

Tabelle 3.1: *Deklaration der Richtungen (E=East, W=West, N=North, S=South, T=Top, B=Bottom, Z=Zero)*

Die Lattice-Boltzmann-Gleichung Gl. 2.7 setzt sich aus folgenden Teilen zusammen:

Kollision

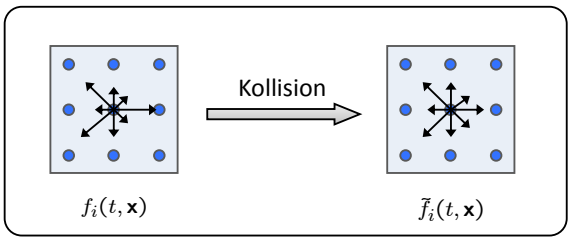


Abbildung 3.1: *Kollision*

$$\tilde{f}_i(t,\mathbf{x}) = f_i(t,\mathbf{x}) + \Omega_i(t,\mathbf{x}) \tag{3.2}$$

In diesem lokalen Schritt wird die Interaktion der Teilchen zum Zeitpunkt t an einem Knoten an Position $\mathbf{x}$ mit Hilfe des Kollisionsoperators Ω modelliert. Für Ω existiert eine Vielzahl von Näherungslösungen. Der bereits in Kap. 2 erwähnte BGK-Ansatz und der MRT-Ansatz [29, 97] gehören zu den bekanntesten und werden im nächsten Abschnitt genauer beschrieben.

Propagation

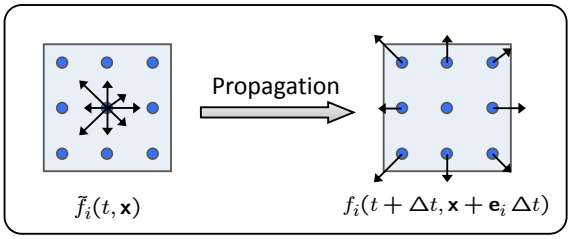


Abbildung 3.2: *Propagation*

$$f_i(t+\Delta t,\mathbf{x}+\mathbf{e}_i\,\Delta t)=\tilde{f}_i(t,\mathbf{x}) \tag{3.3}$$

Diese beschreibt die Advektion der Teilchen. Entsprechend zu $\mathbf{e}_i$ werden die Verteilungen nach der Kollision zum jeweiligen Nachbarknoten kopiert. Dieser nicht lokale Schritt führt während der Berechnung zu einem hohen Kommunikationsaufwand zwischen den Knoten, der bei verteilten Simulationen optimiert werden muss.

3.1 Kollisionsmodelle

BGK-Modell

$$\Omega=-\frac{\Delta t}{\tau}\left(f-f^{eq}\right) \tag{3.4}$$

Bei dem BGK-Ansatz (Gl. 3.4), der zu den Single Relaxation Time (SRT) Modellen gehört, besteht zwischen der Relaxationszeit τ und der kinematischen Viskosität ν folgender Zusammenhang:

$$\tau=3\,\frac{\nu}{c^2}+\frac{1}{2}\,\Delta t \tag{3.5}$$

Die Gleichgewichtsverteilungen ergeben sich für das kompressible Modell [122] zu

$$f_i^{eq}=\mathrm{w}_i\,\rho\left(1+3\,\frac{\mathbf{e}_i\,\mathbf{u}}{c^2}+\frac{9}{2}\frac{(\mathbf{e}_i\,\mathbf{u})^2}{c^4}-\frac{3}{2}\frac{\mathbf{u}^2}{c^2}\right). \tag{3.6}$$

Im sogenannten inkompressiblen Modell [68, 86] ergibt sich f^{eq} zu

$$f_i^{eq}=\mathrm{w}_i\,\rho+\mathrm{w}_i\,\rho_0\left(3\,\frac{\mathbf{e}_i\,\mathbf{u}}{c^2}+\frac{9}{2}\frac{(\mathbf{e}_i\,\mathbf{u})^2}{c^4}-\frac{3}{2}\frac{\mathbf{u}^2}{c^2}\right) \tag{3.7}$$

mit einer konstanten Referenzdichte ρ_0 und der Dichtevariation ρ.

Die Wichtungsfaktoren w_i sind abhängig vom verwendeten Diskretisierungsstern:

$$\mathbf{w}_{d2q9} = \left(\frac{4}{9},\frac{1}{9},\frac{1}{9},\frac{1}{9},\frac{1}{9},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36}\right) \tag{3.8}$$

$$\mathbf{w}_{d3q19} = \left(\frac{1}{3},\frac{1}{18},\frac{1}{18},\frac{1}{18},\frac{1}{18},\frac{1}{18},\frac{1}{18},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36},\frac{1}{36}\right) \tag{3.9}$$

Momentenmodell

Eine weitere Näherung für den Kollisionsoperator Ω ist das ursprünglich von d'Humières entwickelte Momentenmodell [29]:

$$\Omega(t,\mathbf{x}) = \mathsf{M}^{-1}\mathsf{S}\left[\mathsf{M}\mathbf{f}(t,\mathbf{x}) - \mathbf{m}^{eq}(t,\mathbf{x})\right] \tag{3.10}$$

Die Verteilungen $\mathbf{f}(t,\mathbf{x})$ werden hier mit Hilfe einer Transformationsmatrix M in den Momentenraum transfomiert. Dadurch erzeugt man einen neuen Raum bestehend aus 9 ($d2q9$) bzw. 19 ($d3q19$) linear unabhängigen Momenten. Anschließend werden die Momente relaxiert und wieder in den Geschwindigkeitsraum rücktransformiert. Durch die Verwendung verschiedener Relaxationszeiten (Multi Relaxation Time, MRT) für nicht zu konservierende Momente kann letztlich eine höhere Stabilität des Verfahrens und somit eine effizientere Berechnung erzielt werden.

Im Folgenden wird auf die Einzelheiten des $d3q19$-Momentenmodells eingegangen. Die orthogonalen Basisvektoren $\mathbf{\Phi}$ (Abschn. A1) der hier zur Anwendung kommenden Transformationsmatrix M (Abschn. A2) sind in Bezug auf das innere Produkt $\langle \mathbf{\Phi}_a, \mathsf{W}, \mathbf{\Phi}_b \rangle$ mit $\mathsf{W} = diag(\mathrm{w}_i)$ und $a \neq b$ orthogonal zueinander. Im Gegensatz hierzu gilt im Ursprungsmodell $\langle \mathbf{\Phi}_a, \mathbf{\Phi}_b \rangle = 0$ für $a \neq b$ [29].

Die Momente $\mathbf{m} = \mathsf{M}\mathbf{f}$ werden folgendermaßen bezeichnet:

$$\mathbf{m} = \left(\rho, e, \epsilon, j_{x_1}, q_{x_1}, j_{x_2}, q_{x_2}, j_{x_3}, q_{x_3}, 3p_{x_1x_1}, 3\pi_{x_1x_1}, p_{ww}, \pi_{ww}, p_{x_1x_2}, p_{x_2X_3}, p_{x_1x_3}, m_{x_1}, m_{x_2}, m_{x_3}\right)$$

$\mathsf{S} = \{s_{i,i}, i = 0, \ldots, q-1\}$ ist die diagonale Kollisionsmatrix mit den unterschiedlichen Relaxationszeiten. Deren Kollisionsparameter $s_{i,i}$ (die Eigenwerte der Kollisionsmatrix $\mathsf{M}^{-1}\mathsf{S}\mathsf{M}$), die ungleich Null sind, ergeben sich zu:

$$\begin{aligned}
s_{1,1} &= s_e \\
s_{2,2} &= s_\epsilon \\
s_{4,4} &= s_{6,6} = s_{8,8} = s_q \\
s_{10,10} &= s_{12,12} = s_\pi \\
s_{9,9} &= s_{11,11} = s_{13,13} = s_{14,14} = s_{15,15} = -\frac{\Delta t}{\tau} = s_\omega \\
s_{16,16} &= s_{17,17} = s_{18,18} = s_m
\end{aligned} \tag{3.11}$$

In dieser Arbeit werden die Werte für $s_e, s_\epsilon, s_q, s_\pi$ und s_m, die im Allgemeinen frei im Bereich $[-2,0]$ gewählt werden können, zu -1 gesetzt. Die optimalen Werte sind problemabhängig (Geometrie, Randbedingungen, etc.) und können deshalb nicht a priori berechnet werden. Anhaltswerte und ausführliche Studien über die bestmögliche Wahl der Parameter sind in [27, 29, 54, 55] zu finden. Für Stokesströmungen können z. B. die *magic*-Relaxationsparameter [50] verwendet werden, durch die gerade und ungerade Momente unterschiedlich relaxiert werden .

Die Gleichgewichtsmomente $\mathbf{m}^{eq}$ für das $d3q19$-Modell, die ungleich Null sind, berechnen sich wie folgt:

$$\begin{aligned}
m_0^{eq} &= \rho \\
m_1^{eq} &= e^{eq} &&= \rho_0 \left(u_{x_1}^2 + u_{x_2}^2 + u_{x_3}^2\right) \\
m_3^{eq} &= \rho_0\, u_{x_1} \\
m_5^{eq} &= \rho_0\, u_{x_2} \\
m_7^{eq} &= \rho_0\, u_{x_3} \\
m_9^{eq} &= 3 p_{x_1x_1}^{eq} &&= \rho_0 \left(2\, u_{x_1}^2 - u_{x_2}^2 - u_{x_3}^2\right) \\
m_{11}^{eq} &= p_{x_3x_3}^{eq} &&= \rho_0 \left(u_{x_2}^2 - u_{x_3}^2\right) \\
m_{13}^{eq} &= p_{x_1x_2}^{eq} &&= \rho_0\, u_{x_1}\, u_{x_2} \\
m_{14}^{eq} &= p_{x_2x_3}^{eq} &&= \rho_0\, u_{x_2}\, u_{x_3} \\
m_{15}^{eq} &= p_{x_1x_3}^{eq} &&= \rho_0\, u_{x_1}\, u_{x_3}
\end{aligned} \tag{3.12}$$

Während man beim kompressiblen Momentenmodell $\rho_0 = \rho$ verwendet, wird beim inkompressiblen Modell die Referenzdichte ρ_0 meist zu eins gesetzt. Die makroskopischen Werte können beim MRT-Ansatz entweder gemäß Gl. 2.8, 2.9 und 2.11 oder direkt aus den Momenten bestimmt werden (Gl. 3.12 und 3.13).

$$S_{\alpha\beta} = -\left(1 + \frac{s_\omega}{2}\right) \Pi_{\alpha\beta} \tag{3.13}$$

mit

$$\begin{aligned}
\Pi_{x_1x_1} &= \tfrac{1}{3} e + p_{x_1x_1} - \rho_0 u_{x_1}^2 \\
\Pi_{x_2x_2} &= \tfrac{1}{3} e - \tfrac{1}{2} p_{x_1x_1} + \tfrac{1}{2} p_{ww} - \rho_0 u_{x_2}^2 \\
\Pi_{x_3x_3} &= \tfrac{1}{3} e - \tfrac{1}{2} p_{x_1x_1} - \tfrac{1}{2} p_{ww} - \rho_0 u_{x_3}^2 \\
\Pi_{x_1x_2} &= p_{x_1x_2} - \rho_0 u_{x_1} u_{x_2} \\
\Pi_{x_2x_3} &= p_{x_2x_3} - \rho_0 u_{x_2} u_{x_3} \\
\Pi_{x_1x_3} &= p_{x_1x_3} - \rho_0 u_{x_1} u_{x_3}
\end{aligned} \tag{3.14}$$

3.2 Randbedingungen

Bei der Lattice-Boltzmann-Methode existiert keine eindeutige Abbildung der makroskopischen Größen der Navier-Stokes-Gleichungen auf die Wahrscheinlichkeitsverteilungen. Deshalb ist die Implementierung von Randbedingungen im direkten Vergleich aufwändiger. Die in dieser Arbeit verwendeten Randbedingungen werden im Anschluss erläutert. Für Validierungen wird auf die entsprechende Literatur verwiesen.

Haftrandbedingung (no-slip-Rand)

Die Haftrandbedingung fordert, dass sich die Geschwindigkeiten des Fluids und des Randes entsprechen müssen. Bei ortsfesten Rändern sind alle Geschwindigkeitskomponenten gleich Null. Eine Möglichkeit der Implementierung ist das so genannte *Simple-Bounce-Back*-Schema. Der Grundgedanke hierbei ist, dass eine auf die Wand treffende Verteilung in entgegengesetzter Richtung *inv* reflektiert wird und dadurch der übertragene Impuls im zeitlichen Mittel Null ist:

$$f_i(t + \Delta t, \mathbf{x}) = \tilde{f}_{inv}(t, \mathbf{x}) \tag{3.15}$$

Ein großer Nachteil dieser Methode ist, dass implizit davon ausgegangen wird, dass sich der Rand genau in der Mitte zweier Knoten befindet, sodass die Strecke, die von der Verteilung zurückgelegt wird genau $\mathbf{e}_i \Delta t$ entspricht. Bei gekrümmten Rändern wird dadurch das Verfahren auf die Genauigkeit erster Ordnung im Raum reduziert. Eine Weiterentwicklung stellt die *Boundary-Fitting*-Methode [10, 53]

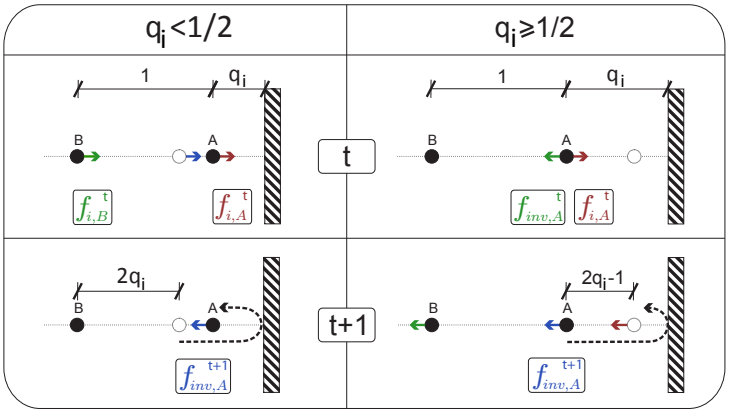


Abbildung 3.3: *Haftrand: Boundary-Fitting-Methode*

dar. Hierbei wird abhängig vom normierten Abstand q_i in Richtung $\mathbf{e}_i$ zum Rand mit $q_i \in]0, |\mathbf{e}_i \Delta x|]$ die zu ermittelnde Verteilung inter- bzw. extrapoliert (Abb. 3.3). Mit der Randgeschwindigkeit $\mathbf{u}_w$ werden sich bewegende Ränder berücksichtigt:

$$0.0 < q_i < 0,5 \quad : \quad f_{inv,A}^{t+1} = (1-2q_i)\, f_{i,B}^{t} + 2q_i\, f_{i,A}^{t} - 2\rho \mathrm{w}_i \frac{\mathbf{e}_i \mathbf{u}_w}{c_s^2} \tag{3.16}$$

$$0,5 \leq q_i \leq 1,0 \quad : \quad f_{inv,A}^{t+1} = \frac{2q_i - 1}{2q_i}\, f_{inv,A}^{t} + \frac{1}{2q_i}\, f_{i,A}^{t} - \rho \mathrm{w}_i \frac{\mathbf{e}_i \mathbf{u}_w}{q_i\, c_s^2} \tag{3.17}$$

Die Methode ist aber aufgrund der Interpolation nicht vollständig masse- und impulserhaltend. Da hier jedoch eine Genauigkeit zweiter Ordnung erreicht wird, ist diese Einschränkung im Allgemeinen vernachlässigbar. Anhand einer Poiseuille-Strömung wird der Unterschied der beiden Methoden insbesondere am Geschwindigkeitsprofil in Wandnähe deutlich (Abb. 3.4 und 3.5).

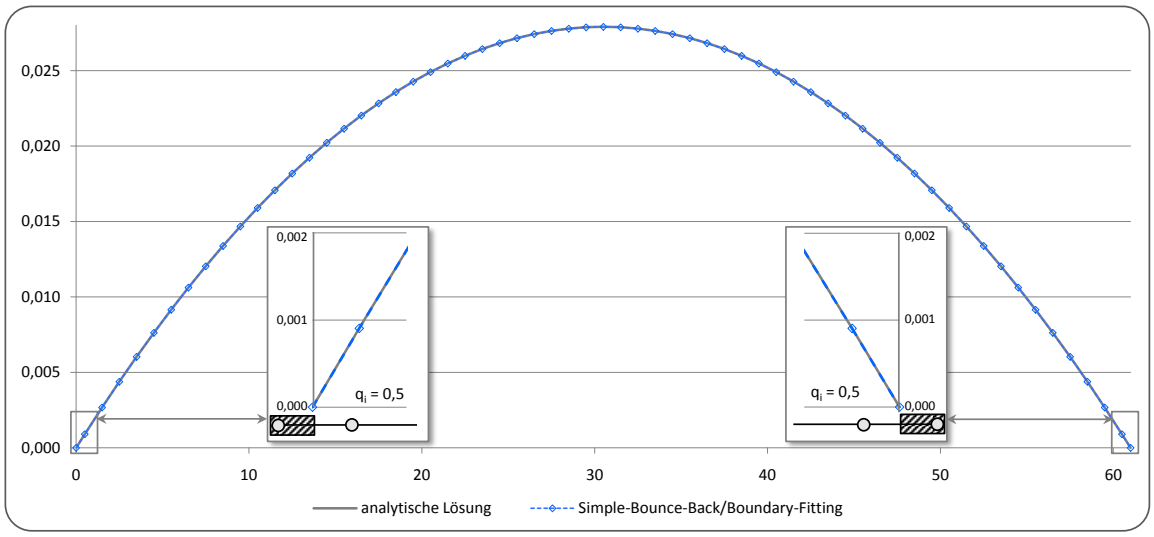


Abbildung 3.4: *Geschwindigkeitsprofil einer Hagen-Poiseuille-Strömung ($q_i = 0,5$)*

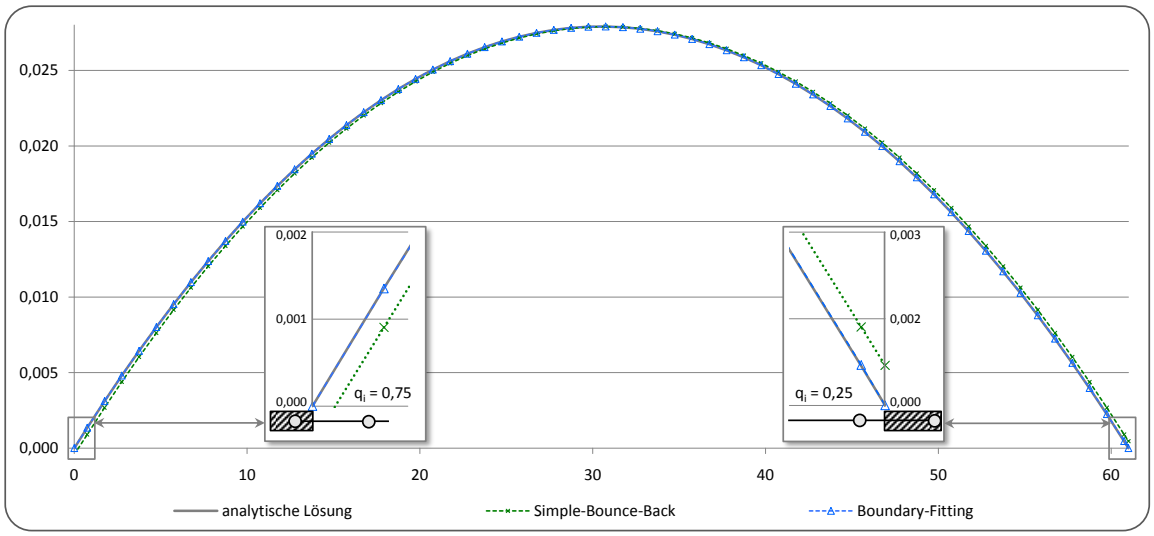


Abbildung 3.5: *Geschwindigkeitsprofil einer Hagen-Poiseuille-Strömung ($q_i \neq 0,5$)*

Rutschrandbedingung (slip-Rand)

Diese Randbedingung kann zum Beispiel bei symmetrischen Strömungsfeldern angewandt werden. Die zum Rand Γ orthogonalen Geschwindigkeitsanteile werden zu Null gesetzt, sodass:

$$\boldsymbol{\eta}_\Gamma \mathbf{u} = 0 \tag{3.18}$$

Erste LB-Umsetzungen waren entweder nur erster Ordnung genau oder konnten nur für gitterparallele Wände verwendet werden [52]. Verbesserte Ansätze sind u. a. in [3] und [149] zu finden.

Ein- und Auslassrandbedingung

Hier wird ein Druck- oder Geschwindigkeitsprofil entlang des Ein- bzw. Ausflusses vorgegeben. Weitere zur Bestimmung der Gleichgewichtsverteilungen notwendige makroskopische Werte erhält man entweder direkt vom orthogonal zum Rand liegenden Nachbarknoten oder durch Extrapolation mehrerer orthogonal zum Rand liegender Fluidnachbarknoten [154].

- **Geschwindigkeitsrand** [98]
  Da sich dieser aus mathematischer Sicht nicht vom Haftrand unterscheidet, können Gl. 3.16 und 3.17 analog angewendet werden, wobei die Wandgeschwindigkeit der Einflussgeschwindigkeit entspricht.
- **Druckrand** [153]

  $$f_i(t+\Delta t, \mathbf{x}) = -\tilde{f}_{inv}(t, \mathbf{x}) + f^{eq}_{inv}(\mathbf{u}, \rho_{BC}) + f^{eq}_i(\mathbf{u}, \rho_{BC}) \tag{3.19}$$

  mit

  $$\mathbf{u} = \mathbf{u}\left(t + \frac{\Delta t}{2}, \mathbf{x} + \frac{\mathbf{e}_{inv}\,\Delta t}{2}\right)$$

  Bei dieser Randbedingung können es beim Auftreten von Wirbelablösungen im Randbereich numerische Instabilitäten auftreten, die durch die Vernachlässigung der inversen Verteilungsterme in Gl. 3.19 behoben werden können.

Periodischer Rand

Ein periodisches Rechengebiet wird am jeweiligen Rand mit den Werten des gegenüberliegenden Randes fortgesetzt. Hierzu werden die nach der Propagation fehlenden ein- bzw. ausgehenden Verteilungen entsprechend kopiert, z. B. für ein x_1-periodisches 2-D-Gebiet mit der Länge L_{x_1}:

$$f_{(NE,E,SE)}\left(t,(0,x_2)^T\right) = \tilde{f}_{(NE,E,SE)}\left(t,(L_{x_1},x_2)^T\right); \tag{3.20}$$

$$f_{(NW,W,SW)}\left(t,(L_{x_1},x_2)^T\right) = \tilde{f}_{(NW,W,SW)}\left(t,(0,x_2)^T\right); \tag{3.21}$$

Volumenkräfte (forcing)

Zu den wichtigsten externen Kräften gehört die Schwergewichtskraft. Um derartige Volumenkräfte $\mathbf{F}$ zu berücksichtigen, wird in jedem Zeitschritt der Zusatzterm

$$\delta f_i = \mathrm{w}_i \frac{1}{c_s^2} \rho \, \Delta t \, \mathbf{e}_i \, \mathbf{F} \tag{3.22}$$

auf die Verteilungen addiert.

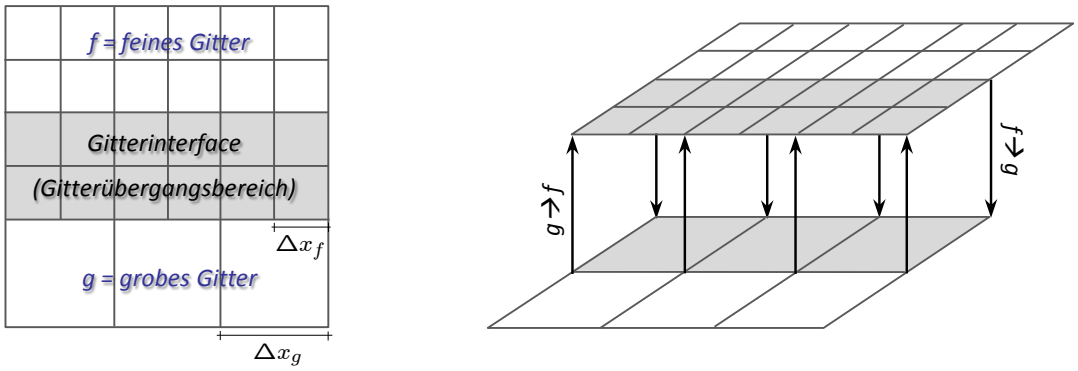
Im MRT-Modell kann die Kraft alternativ direkt auf die Impulsmomente j_{x_i} (vgl. Gl. 3.1) addiert werden

$$\delta j_{x_i} = \rho \, \Delta t \, F_{x_i} \tag{3.23}$$

In [12, 49, 53, 67, 96] wurden Untersuchungen hinsichtlich der Berücksichtigung von Volumenkräften im LB-Kontext durchgeführt. Eine Übersicht und ein Vergleich der unterschiedlichen Ansätze mit dem Ergebnis, dass diese untereinander vergleichbar sind, ist in [54] zu finden.

3.3 Nicht-uniforme Gitter

Die in dieser Arbeit besprochenen nicht-uniformen, strukturierten, kartesischen Gitter werden aus verschiedenen Gründen verwendet. Zwei wesentliche Aspekte sind die Minimierung des Speicherbedarfs und die Verkürzung der Rechenzeit. Selbst im Zeitalter von Supercomputern sind dadurch effiziente Berechnungen von komplexen Problemen oftmals überhaupt erst möglich. Aus numerischer Sicht lassen sich mit höheren Gebietsauflösungen hohe Gradienten und Wirbelstrukturen (genauer) auflösen. Aber auch die Abbildung von Geometrien lässt sich dadurch optimieren.



(a) *Bezeichnungen* ***(b)*** *Entkoppelte Darstellung*

Abbildung 3.6: *Überlappendes Gitterinterface mit Skalierungsrichtungen für 2-D-Gitter*

Das derzeit fast ausschließlich auf uniformen, kartesischen Gittern angewandte Lattice-Boltzmann-Verfahren lässt sich für nicht-uniforme Gitter erweitern [22, 23, 37, 166]. Ein hierbei typisches Gitterinterface ist in Abb. 3.6 dargestellt. Die Überlappung ist notwendig, da fehlende Verteilungen eines Knotens mit den Informationen seines korrespondierenden Nachbarknotens des anderen Gitterlevel berechnet werden.

Folgende Bedingungen müssen am Übergangsbereich erfüllt sein:

- Massenerhaltung
$$\sum_i f_{i,g}(t,\mathbf{x}) = \sum_i f_{i,f}(t,\mathbf{x}) \tag{3.24}$$
- Impulserhaltung
$$\sum_i \mathbf{e}_i f_{i,g}(t,\mathbf{x}) = \sum_i \mathbf{e}_i f_{i,f}(t,\mathbf{x}) \tag{3.25}$$
- Kontinuität der Spannungen
$$S_{\alpha\beta} = -\left(1 - \frac{\Delta t_l}{2\,\tau_l}\right) \sum_i e_{\alpha i}\, e_{\beta i} \left(f_{i,l}(t,\mathbf{x}) - f_{i,l}^{eq}(t,\mathbf{x})\right) \tag{3.26}$$

Die Schallgeschwindigkeit soll im gesamten System konstant sein. Somit ergibt sich, bei konstanter Viskosität, die Relaxationszeit (vgl. Gl. 3.5) auf unterschiedlichen Gitterleveln l zu [22]:

$$\tau_l = 3\frac{\nu}{c^2} + \frac{1}{2}\Delta t_l \tag{3.27}$$

► **Skalierung für das Standard-BGK-Modell**

Während die Gleichgewichtsverteilungen f_i^{eq} unabhängig vom Gitterlevel sind:

$$f_i^{eq}(t,\mathbf{x}) = f_{i,g}^{eq}(t,\mathbf{x}) = f_{i,f}^{eq}(t,\mathbf{x}) \tag{3.28}$$

müssen die Nichtgleichgewichtsverteilungen $f_{i,l}^{neq}$ zwischen verschiedenen Verfeinerungsebenen vor der Kollision skaliert werden, um die Kontinuität des Spannungstensors zu gewährleisten. Aus

$$\text{grob} \rightarrow \text{fein} \quad : \quad f_{i,f}(t,\mathbf{x}) = f_i^{eq}(t,\mathbf{x}) + s_{g\rightarrow f}\, f_{i,g}^{neq}(t,\mathbf{x}) \tag{3.29}$$

$$\text{fein} \rightarrow \text{grob} \quad : \quad f_{i,g}(t,\mathbf{x}) = f_i^{eq}(t,\mathbf{x}) + s_{f\rightarrow g}\, f_{i,f}^{neq}(t,\mathbf{x}) \tag{3.30}$$

mit

$$f_{i,l}^{neq}(t,\mathbf{x}) \;=\; f_{i,l}(t,\mathbf{x}) \;-\; f_i^{eq}(t,\mathbf{x}) \tag{3.31}$$

folgt

$$s_{g\rightarrow f} \;=\; \frac{f_{i,f}^{neq}(t,\mathbf{x})}{f_{i,g}^{neq}(t,\mathbf{x})} = \frac{6\nu + \Delta t_f}{6\nu + \Delta t_g} \tag{3.32}$$

$$s_{f\rightarrow g} \;=\; \frac{1}{s_{g\rightarrow f}} \tag{3.33}$$

► **Skalierung beim Momentenmodell**

Analog zum BGK-Ansatz sind beim Momentenmodell die Gleichgewichtsmomente gitterlevelunabhängig und die Nichtgleichgewichtsmomente $\mathbf{m}^{neq}$ müssen zwischen den Verfeinerungsstufen entsprechend skaliert werden:

$$m_{i,g}^{neq}(t,\mathbf{x}) = \frac{s_{f,i,i}}{s_{g,i,i}}\,\frac{\Delta t_g}{\Delta t_f}\, m_{i,f}^{neq}(t,\mathbf{x}) \tag{3.34}$$

Die Relaxationsparameter $s_e, s_\epsilon, s_q, s_\pi, s_m$ sind levelweise frei im Bereich $[-2,0]$ wählbar und werden hier zu -1 gesetzt.

> ⚠ *Für eine konsistente Skalierung müssen sich die Verteilungen/Momente beider Gitterlevel in einem konsistenten Zustand befinden (post collision und post propagation).*

Eine weitere Bedingung bezüglich identischer Mach- und Reynoldszahl in allen Gitterleveln ist die Verwendung eines gestaffelten Zeitschrittverfahrens (nested time-stepping) [37]. Im feinen Gitter werden dadurch für jeden groben Zeitschritt genau $2^{l_f - l_g}$ Kollisionen und Propagationen durchgeführt (Abb. 3.7).

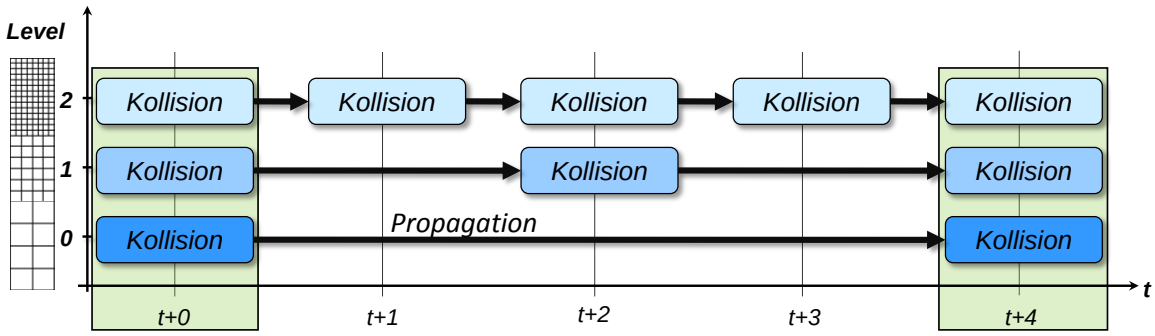


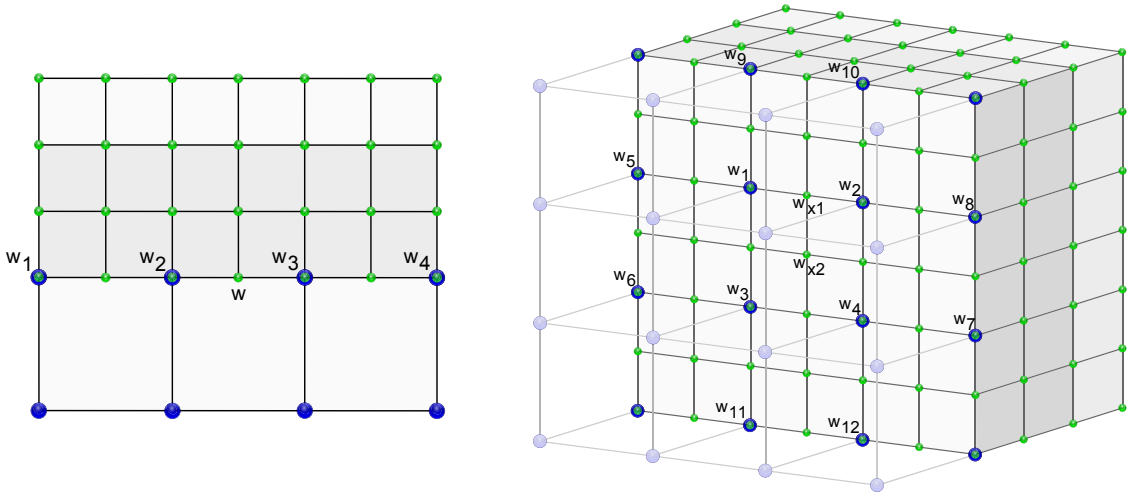
Abbildung 3.7: *Gestaffeltes Zeitschrittverfahren (nested time-stepping)*

Ein anderer Ansatz für LB-Berechnungen auf nicht-uniformen Gittern, bei dem die Machzahl auf dem feinen Gitter reduziert wird, um eine höhere Konvergenzrate bezüglich der inkompressiblen Navier-Stokes-Gleichungen zu erreichen, ist in [126] zu finden. Dieser Ansatz erfordert im feinen Level $4^{l_f - l_g}$ Relaxationen und Advektionen während eines groben Zeitschritts.

Um den algorithmischen Aufwand in Grenzen zu halten, kommen in dieser Arbeit geglättete, baumstrukturierte Gitter zum Einsatz, d. h. benachbarte Knoten unterscheiden sich um maximal einen Gitterlevel. Aufgrund der unterschiedlichen Knotenabstände Δx_l und Zeitschrittintervalle Δt_l muss zur Berechnungszeit interpoliert werden:

- **räumlich**

  Um die fehlenden Informationen der hängenden Knoten, die im Übergangsbereich am feinen Interface auftreten (Abb. 3.8), zu ermitteln bedarf es laut [22] mindestens einer quadratischen Interpolation. Zur Vermeidung dadurch auftretender asymmetrische Effekte wird an diesen Knoten eine kubische Interpolation durchgeführt (Gl. 3.35 und 3.36). Bei unvollständigen Sternen (z. B. an Gebietsrändern) wird gemäß [22] inter- bzw. extrapoliert.



(a) 2-D *(b)* 3-D

Abbildung 3.8: *Interpolationsnachbarn für hängende Knoten*

$$\text{2-D}: \quad \mathrm{w} = \frac{9}{16}(\mathrm{w}_2+\mathrm{w}_3) - \frac{1}{16}(\mathrm{w}_1+\mathrm{w}_4) \tag{3.35}$$

$$\text{3-D}: \quad \mathrm{w}_{x1} = \frac{9}{16}(\mathrm{w}_1+\mathrm{w}_2) - \frac{1}{16}(\mathrm{w}_5+\mathrm{w}_8) \tag{3.36}$$

$$\mathrm{w}_{x2} = \frac{5}{16}\sum_{k=1}^{4}\mathrm{w}_k - \frac{1}{32}\sum_{k=5}^{12}\mathrm{w}_k \tag{3.37}$$

► **zeitlich**

Aufgrund des gestaffelten Zeitschrittverfahrens müssen Zeitinterpolationen für die feinen Zwi-

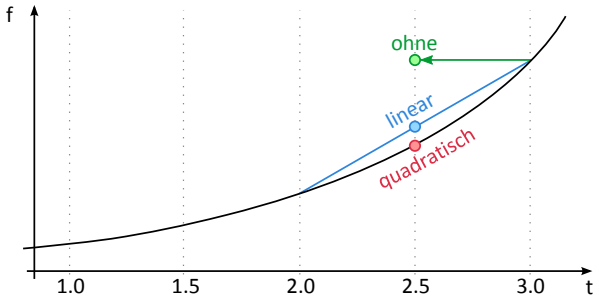


Abbildung 3.9: *Vergleich der Zeitinterpolationen*

schenschritte durchgeführt werden. Nach [22] ist es hinreichend, für stationäre Strömungsprobleme die Werte der groben Knoten zu übernehmen. Für instationäre Strömungen hingegen sollte mindestens eine lineare Interpolation durchgeführt werden:

$$f_i\left(t+\frac{1}{2}\Delta t, \mathbf{x}\right) = \frac{1}{2}\left(f_i(t,\mathbf{x}) + f_i(t+\Delta t,\mathbf{x})\right) \tag{3.38}$$

Da höherwertige Interpolationsverfahren nur eine geringfügige Verbesserung liefern [22], werden in dieser Arbeit im Gitterübergangsbereich ausschließlich lineare Zeitinterpolationen verwendet (Abb. 3.9).

4 Lattice-Boltzmann-Methode für Mehrphasenprobleme

4.1 Allgemeines

In dieser Arbeit wird bei Mehrphasenproblemen von nicht-mischbaren Zweiphasensystemen ausgegangen. Charakteristisch sind die beweglichen und frei deformierbaren Phasengrenzen, an denen sich die Stoffeigenschaften sprunghaft ändern. Ein weiteres Kennzeichen ist die Oberflächenspannung an gekrümmten Phasenrändern, die sich in dem Bestreben eine möglichst kleine Oberfläche zu bilden äußert.

Hierfür existiert eine Vielzahl von Verfahren, die auf der direkten Diskretisierung der Navier-Stokes-Gleichungen beruhen. Zur Behandlung der Phasengrenze für nicht-mischbare Phasen seien hier die Volume of Fluid (VOF) [65, 77], die Level-Set- [116, 137, 147] und Partikel-Level-Set Methoden [35] genannt.

Die besondere Schwierigkeit von Zweiphasenproblemen rührt von den scharfen, sich bewegenden Fronten her, an denen Druck und Geschwindigkeiten Unstetigkeiten aufweisen können. Die zahlreichen Varianten zur Beschreibung der Phasengrenze und ihrer Bewegung basieren häufig auf ähnlichen Diskretisierungen der Grundgleichungen. Die Verfahren zur Simulation von Mehrphasenproblemen lassen sich im Wesentlichen in zwei Kategorien einteilen: die Surface-Tracking- und die Surface-Capturing-Methoden (ein Überblick wird in [80] gegeben).

Surface-Tracking-Verfahren behandeln die Phasengrenze explizit als Unstetigkeit. Sie wird durch Markierungspartikel dargestellt, die durch eine Interpolationskurve verbunden werden. Diese Markierungspartikel werden dann mittels einer Lagrange-Beschreibung propagiert und anschließend so neu verteilt, dass die Phasengrenze möglichst gut aufgelöst wird [129].

Surface-Capturing-Verfahren beziehen sich implizit auf die Phasengrenze. Die genaue Position der Phasengrenze wird zur Diskretisierung der Grundgleichungen nicht benötigt, da sie spezielle Funktionen verwenden, um die Phasenverteilung zu beschreiben. Beispiele hierfür sind das Fluidvolumenverfahren von Hirt und Nichols [77] sowie der Level-Set-Ansatz von Osher und Sethian [117]. Diese Verfahren eignen sich u. U. besser als die Surface-Tracking-Verfahren zur Behandlung komplizierter Phasengrenzen und erlauben auch die Beschreibung von Aufspaltung und Wiedervereinigung derselben, wie es z. B. beim Ablösen und Wiederanlegen eines Tropfens der Fall ist.

Bei der VOF-Methode wird die Phasenverteilung durch die Volumenanteile jeder Phase in einer Zelle beschrieben. Für Zellen, die von der Phasengrenze geschnitten werden, muss dieser Volumenanteil also irgendwo zwischen Null und Eins liegen. Demgegenüber wird bei der Level-Set-Formulierung die Phasengrenze implizit als die Menge der Nullstellen einer glatten Funktion f beschrieben. Auf diese Weise können Aufbrechen und Wiederverschmelzen der Front bequem beschrieben werden, da die Level-Set-Funktion auch bei Topologieänderungen wohldefiniert bleibt. In neueren Veröffentlichungen [147, 117] wird der vorzeichenbehaftete Abstand von der Phasengrenze als eine solche Funktion verwendet, hiermit lässt sich die Krümmung der Phasengrenze sehr genau berechnen. Ad-

vektionsschemata höherer Ordnung für die VOF-Methode sind nicht trivial, entsprechende Verfahren sind in [109, 121, 127] zu finden. Einfacher ist die Advektion der Phasengrenze in der Level-Set-Formulierung, da hier nur Advektion der glatten Level-Set-Funktion durchgeführt werden muss, was z. B. mit einem essentially-nonoscillator(ENO)-Schema zweiter Ordnung [147] geschieht.

Neuere Entwicklungen kombinieren den Phasenfeldansatz und die explizite Darstellung der Phasengrenzfläche [125], indem eine Phasenfeldfunktion zur Advektion der Geometrie verwandt wird, dann aber für die Implementierung der Randbedingungen das Interface explizit lokal rekonstruiert wird.

Adaptive Rechnungen und Verfeinerungskriterien für die Level-Set-Methode sind in [147] zu finden, wobei ein hierarchisches und strukturiertes Gitter verwendet wird. Kombinierte Fehlerschätzer zur Steuerung der Orts- und Zeitschrittweite sind in [9] beschrieben. Verfahren auf unstrukturierten Gittern mit Mehrgitterlösern für zweidimensionale Probleme in einfachen Geometrien wurden in [56] entwickelt.

Direkte Vergleiche zwischen der LB-Methode und der VOF-Methode am Beispiel eines Tropfenstoßes und aufsteigender Luftblasen sind in [130, 133] zu finden. Schon bei dieser einfachen Problemgeometrie erweist sich die LB-Methode bei vergleichbarer Ergebnisqualität als sehr effizient und erlaubt eine Reduktion der benötigten Rechenzeit um fast eine Größenordnung (jeweils auf uniformen Gittern).

Zur Simulation von Mehrphasenströmungen im Lattice-Boltzmann-Kontext existieren im Wesentlichen drei verschiedene Ansätze: Das Rothman-Keller(RK)-Modell [63, 66, 128], das Shan-Chen-Modell [138] und das Free-Energy-Modell [148]. Der Vorteil des RK-Modells ist die Möglichkeit, die Oberflächenspannung, das Dichte- und Viskositätsverhältnis unabhängig voneinander einzustellen. Beim Shan-Chen-Modell besteht zwar die Möglichkeit hoher Dichteverhältnisse, aber die Oberflächenspannung, das Dichte- und Viskositätsverhältnis sind nicht unabhängig voneinander konfigurierbar [132]. Des Weiteren müssen einige Parameter durch numerische Experimente ermittelt werden. Der größte Nachteil des Free-Energy-Modells ist, dass der viskose Term in den aus der Analyse resultierenden Navier-Stokes-Gleichungen nicht Galilei-Invariant ist [100]. Eine detaillierte Analyse des RK-Modells ist in [90] durchgeführt.

In dieser Arbeit wurde zunächst eine Variante des Shan-Chan-Modells implementiert. Aufgrund numerischer Instabilitäten wurde letztlich eine eigenständige Erweiterung des RK-Modells entwickelt.

4.2 Erweitertes Shan-Chen-Modell

Zu Beginn der Arbeit wurde das in [70] vorgestellte, erweiterte Potentialmodell für nicht-uniforme Gitter implementiert. Bei potentialbasierten Modellen wird für Phasen mit unterschiedlicher Dichte eine Potentialfunktion $\Psi(\rho)$ eingeführt, durch die an Phasengrenzen aufgrund des lokalen Dichtegradienten Kräfte induziert werden, die eine Vermischung der einzelnen Phasen verhindern. Diese Funktion ist so konstruiert, dass sie in einem nicht-idealen Gasgesetz (van der Waals) resultiert und dadurch für bestimmte makroskopische Drücke zwei verschiedene Dichtewerte aufweist (Abb. 4.1).

Bei dem Shan-Chen-Modell kann das Potential V folgendermaßen definiert werden [132]:

$$V(\mathbf{x},t) = G\Psi(t,\mathbf{x})^2 \tag{4.1}$$

Die zwischenmolekularen Kräfte ergeben sich zu:

$$\mathbf{F}(t,\mathbf{x}) = -\nabla V(t,\mathbf{x}) = -2G\,\Psi(t,\mathbf{x})\,\nabla\Psi(t,\mathbf{x}) \tag{4.2}$$

und das Gasgesetz lautet:

$$p = c^2 \left[\frac{1}{3}\rho + G\,\Psi(t,\mathbf{x})^2 \right] \tag{4.3}$$

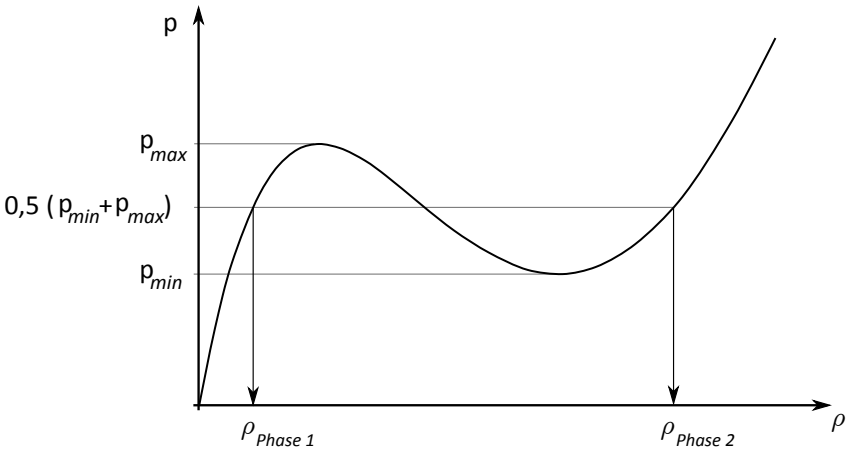


Abbildung 4.1: *Nicht-ideales Gasgesetz $p(\rho)$*

Die Kraft wird über die Gleichgewichtsverteilungen in das Verfahren eingeführt, wobei die Geschwindigkeit bei der Berechnung der Gleichgewichtsverteilungen umdefiniert wird:

$$\mathbf{u}' = \left(\tau - \frac{\Delta t}{2}\right)\frac{\mathbf{F}}{\rho} + \mathbf{u} \tag{4.4}$$

Der große Nachteil dieses Verfahrens ist jedoch, dass die Oberflächenspannung sowohl vom Gitterabstand Δx als auch vom Dichteverhältnis abhängt und lediglich im Limes kleiner Viskosität von dieser unabhängig justiert werden kann.

Deshalb wurde ein neuer Kraftansatz [70] implementiert, bei dem die Oberflächenspannung über einen eigenen Parameter κ gesteuert werden kann.

$$\mathbf{F} = -\nabla V' - b\,\rho\,RT\,\chi\,\nabla ln\left(\rho^2\chi\right) = -\nabla V \tag{4.5}$$

mit

$$\chi = 1 + 0{,}625\,(b\,\rho) + 0{,}2869\,(b\,\rho)^2 + 0{,}1103\,(b\,\rho)^3 + \ldots \tag{4.6}$$

Das zu verwendende Kraftpotential V' lautet:

$$V' = -2a\rho - \kappa\triangle\rho \tag{4.7}$$

Durch mathematische Umformung lässt sich zeigen, dass sich die gesuchte Potentialfunktion V in Gl. 4.5 zu einem Polynom vereinfachen lässt:

$$V = A_1\rho + A_2\rho^2 + A_3\rho^3 + \kappa\triangle\rho \tag{4.8}$$

Das Gasgesetz lautet nun:

$$p = \frac{1}{3}\rho + A_1\rho + A_2\rho^2 + A_3\rho^3 \tag{4.9}$$

Die Kräfte selbst werden als Volumenkraft (Forcing) auf die Verteilungen addiert:

$$f_i\,(t+\Delta t,\mathbf{x}+\mathbf{e}_i\Delta t) - f_i(t,\mathbf{x}) = \frac{-\Delta t}{\tau}\left(f_i(t,\mathbf{x}) - f_i^{eq}(t,\mathbf{x})\right) + 4\,\Delta t\,f^{eq}(t,\mathbf{x})\,\frac{\mathbf{F}\cdot(\mathbf{e}_i-\mathbf{u})}{\mathbf{u}^2} \tag{4.10}$$

Für die Implementierung wurde $\mathbf{F}$ in zwei Terme gesplittet:

$$\mathbf{F} = \mathbf{F}_1 + \mathbf{F}_2 \quad \text{mit} \quad \mathbf{F}_1 = \nabla\left(A_1\rho + A_2\rho^2 + A_3\rho^3\right) \quad \text{und} \quad \mathbf{F}_2 = -\kappa\nabla\triangle\rho$$

$\mathbf{F}_1$ ist für das nicht-ideale Gasgesetz und $\mathbf{F}_2$ für die Oberflächenspannung verantwortlich. Durch die Bildung von verschiedenen Finite-Differenzen-Sternen (eine Variante ist in Abb. 4.2 und Abb. 4.3 zu sehen) wurden die Kraftanteile numerisch bestimmt.

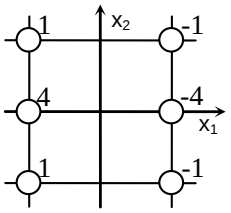


Abbildung 4.2: *Diskretisierungsstern für* F_{1,x_1}

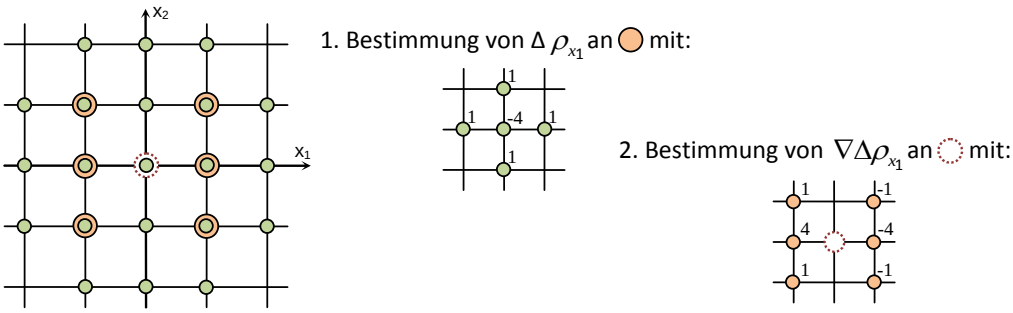


Abbildung 4.3: *Diskretisierungsstern für* F_{2,x_1}

Die Implementierung des erweiterten Potentialgesetzes gestaltete sich gerade im Übergangsbereich bei nicht-uniformen Gittern als schwierig. Das Verfahren führte sehr große Störungen ein und war relativ instabil. Vermutlich führt die Bestimmung des Kraftanteils $\mathbf{F}_1$ (Ableitungen erster Ordnung) durch asymmetrische Finite-Differenzen im Übergangsbereich der Gitter zu numerischen Fehlern höherer Ordnung, die die Kraftanteile aus $\mathbf{F}_2$ dominieren. Ein Beispiel, wie sehr die numerischen Effekte das Verfahren dominiert, kann mit der numerischen Bestimmung der Kraft des Shan-Chen-Modells gegeben werden: Wird die Kraft (Gl. 4.2) numerisch aus dem Term $-2G\,\Psi(t,\mathbf{x})\,\nabla\Psi(t,\mathbf{x})$ gebildet, läuft das Verfahren stabil, verwendet man hingegen $-\nabla G\Psi(t,\mathbf{x})^2$, ist das Verfahren instabil. Formale Stabilitätsanalysen und weitere Diskretisierungssterne (bis hin zu sechster Ordnung) sind in [103] zu finden.

4.3 Erweitertes Rothmann-Keller Modell

Das ursprünglich von Rothmann und Keller für Gittergase entwickelte Zweiphasenmodell [128] wurde von Gunstensen auf das Lattice-Boltzmann-Verfahren übertragen [66] und von Grunau für Phasen unterschiedlicher Dichte und Viskosität erweitert [63]. Zu den Vorteilen des Modells gehört, dass die Oberflächenspannung sowie das Dichte- und Viskositätsverhältnis unabhängig voneinander eingestellt werden können. Gegenüber anderen LB-Ansätzen wird zudem bei der Kollision lokale Massenerhaltung der einzelnen Phasen sowie die lokale Impulserhaltung für die Summe der Teilimpulse der einzelnen Phasen gewährleistet.

4.3.1 Grundlagen

Die Theorie basiert auf einem Phasenfeldansatz. Der entscheidende Unterschied des erweiterten Modells zum Ursprungsmodell ist, dass hier das Strömungsfeld vom Phasenfeld entkoppelt wird. Hierzu wird zunächst der Ordnungsparameter ϕ eingeführt, der über das blaue (ρ_b) und rote (ρ_r) Dichtefeld definiert wird:

$$\phi = \frac{\rho_r - \rho_b}{\rho_r + \rho_b} \tag{4.11}$$

Am Übergang von der roten zu der blauen Phase befindet sich das sogenannte Phaseninterface

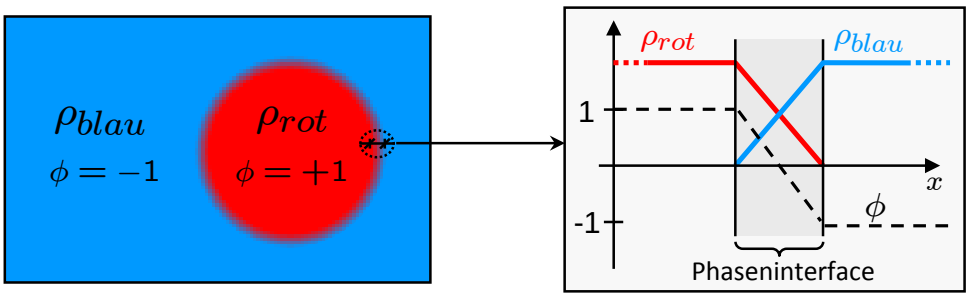


Abbildung 4.4: *Phaseninterface*

(Abb. 4.4). Um dort den Gradienten des Phasenfeldes $\mathbf{C_l}$ zu bestimmen, wird die Verbesserung von Kehrwald [90] verwendet:

$$\mathbf{C}_l(t,\mathbf{x}) = \frac{3}{2\,c^2\,\Delta t_l} \sum_i \mathrm{w}_i\,\mathbf{e}_i\,\phi\,(t,\mathbf{x}+\mathbf{e}_i\Delta t_l) \tag{4.12}$$

sowie dessen Normierung

$$n_{l,\alpha} = \frac{C_{l,\alpha}}{|\mathbf{C}_l|}. \tag{4.13}$$

Die Berechnung des Phasenfeldgradienten $\mathbf{C}_l$ hat zur Folge, dass die Kollision nicht mehr lokal stattfindet, da hierfür Informationen der Nachbarknoten benötigt werden. Dies spielt vor allem bei der späteren Parallelisierung des Berechnungskerns eine wichtige Rolle.

Strömungsfeld

Außerhalb des Phaseninterfacebereichs finden für das Strömungsfeld die Gleichungen des Einphasenansatzes Anwendung. Innerhalb müssen diese zur Berücksichtigung der zweiphasenspezifischen

Eigenschaften angepasst werden. Um die Oberflächenspannung σ zu berücksichtigen, wird die Kollision mit Hilfe einer vom Phasenfeldgradienten abhängigen Störung erweitert, die in die Gleichgewichtsmomente einfließt:

$$\Omega_l(t,\mathbf{x}) = \mathsf{M}^{-1}\mathsf{S}_l\left[\mathsf{M}\,\mathbf{f}(t,\mathbf{x}) - \mathbf{m}'^{eq,l}(t,\mathbf{x})\right] \tag{4.14}$$

Für Zweiphasenprobleme werden in dieser Arbeit die Komponenten $s_e, s_\varepsilon, s_q, s_\pi$ und s_m von S_l zu -1 gesetzt (vgl. Abschn. 3.1). Für Berechnungen auf nicht-uniformen Gittern wird hier, abweichend zum Einphasenmodell, dieselbe Skalierungsvorschrift, wie für $s_{l,\omega}$ angewendet (vgl. Abschn. 3.3):

$$\frac{s_{g,e}}{s_{f,e}} = \frac{s_{g,\varepsilon}}{s_{f,\varepsilon}} = \frac{s_{g,q}}{s_{f,q}} = \frac{s_{g,\pi}}{s_{f,\pi}} = \frac{s_{g,m}}{s_{f,m}} = \frac{s_{g,\omega}}{s_{f,\omega}} \tag{4.15}$$

Die Gleichgewichtsmomente des Einphasenmodells $\mathbf{m}^{eq}$ (Gl. 3.12) werden folgendermaßen erweitert:

$$\begin{aligned}
m_1'^{eq} &= m_1^{eq} - 2\sigma\,|\mathbf{C}_l|\,(n_{l,x_1}^2 + n_{l,x_2}^2 + n_{l,x_3}^2) = m_1^{eq} - 2\sigma\,|\mathbf{C}_l| \\
m_9'^{eq} &= m_9^{eq} + \sigma\,|\mathbf{C}_l|\,(2\,n_{l,x_1}^2 - n_{l,x_2}^2 - n_{l,x_3}^2) \\
m_{11}'^{eq} &= m_{11}^{eq} + \sigma\,|\mathbf{C}_l|\,(n_{l,x_2}^2 - n_{l,x_3}^2) \\
m_{13}'^{eq} &= m_{13}^{eq} + \sigma\,|\mathbf{C}_l|\,(n_{l,x_1}\,n_{l,x_2}) \\
m_{14}'^{eq} &= m_{14}^{eq} + \sigma\,|\mathbf{C}_l|\,(n_{l,x_2}\,n_{l,x_3}) \\
m_{15}'^{eq} &= m_{15}^{eq} + \sigma\,|\mathbf{C}_l|\,(n_{l,x_1}\,n_{l,x_3})
\end{aligned} \tag{4.16}$$

Das BGK-Modell verwendet folgende Gleichgewichtsverteilungen mit f^{eq} aus Gl. 3.12:

$$f_i'^{eq} = f_i^{eq} + \frac{9}{2}\sigma \mathrm{w}_i |\mathbf{C}_l| \left(\frac{\mathbf{e}_i \cdot \mathbf{C}_l}{|\mathbf{C}_l|^2} - |\mathbf{e}_i|^2 + \frac{Dimension}{3} - \frac{1}{3} \right) \tag{4.17}$$

Für den Drucktensor ergeben sich aufgrund der Oberflächenspannung (OS) folgende Erweiterungen:

$$\begin{aligned}
P_{x_1x_1}^{OS} &= \;\;2\sigma\,|\mathbf{C}_l|\,(n_{l,x_2}^2 + n_{l,x_3}^2) \\
P_{x_2x_2}^{OS} &= \;\;2\sigma\,|\mathbf{C}_l|\,(n_{l,x_1}^2 + n_{l,x_3}^2) \\
P_{x_3x_3}^{OS} &= \;\;2\sigma\,|\mathbf{C}_l|\,(n_{l,x_1}^2 + n_{l,x_2}^2) \\
P_{x_1x_2}^{OS} &= -2\sigma\,|\mathbf{C}_l|\,(n_{l,x_1}\,n_{l,x_2}) \\
P_{x_2x_3}^{OS} &= -2\sigma\,|\mathbf{C}_l|\,(n_{l,x_2}\,n_{l,x_3}) \\
P_{x_1x_3}^{OS} &= -2\sigma\,|\mathbf{C}_l|\,(n_{l,x_1}\,n_{l,x_3})
\end{aligned} \tag{4.18}$$

Für Phasen unterschiedlicher Dichte wird im Unterschied zu [63] das Temperaturmodell von Tölke [156] genutzt. Jeder Phase wird dabei eine Temperatur $T_{r/b}$ zugewiesen. Diese bestimmt die Schallgeschwindigkeit der jeweiligen Phase und durch die Beziehung zum Druck p die Dichten der jeweiligen Phase:

$$p = \rho_r(t,\mathbf{x})\;T_r(t,\mathbf{x}) = \rho_b(t,\mathbf{x})\;T_b(t,\mathbf{x}) \tag{4.19}$$

Da die Temperatur eine phasenabhängige Konstante ist und Zwischenwerte im Phasenübergangsbereich mit

$$T = \frac{1}{2}\big(T_r(1+\phi) + T_b(1-\phi)\big) \tag{4.20}$$

interpoliert werden, benötigt man zur Laufzeit kein zusätzliches Temperaturfeld. Um negative Momente/Verteilungen zu vermeiden und somit die Stabilität des Verfahrens zu gewährleisten, muss sich T im Bereich von $[0, \frac{5}{9}]$ befinden. Dadurch wird der maximale Dichteunterschied der beiden Phasen, der effektiv verwendet werden kann, auf ein Verhältnis von 1:50 beschränkt. Ist man an größeren Dichteverhältnissen interessiert, so kann z. B. der in [151] beschriebene Ansatz verwendet werden, bei dem die Entwicklung des Phaseninterfaces mit Hilfe einer Level-Set-Methode modelliert wird.

Die Gleichgewichtsmomente des Einphasenmodells m_i^{eq} werden bei Verwendung des Temperaturfeldes durch $m_i^{eq,T}$ ersetzt:

$$\begin{aligned}
m_0^{eq,T} &= \rho \\
m_1^{eq,T} &= e^{eq,T} &&= \rho_0 \left(2c^2 \left(3T-1\right) + u_{x_1}^2 + u_{x_2}^2 + u_{x_3}^2\right) \\
m_2^{eq,T} &= \epsilon^{eq,T} &&= \tfrac{1}{5}\,\rho_0\, c^4 \left(5 - 9\,T\right) \\
m_3^{eq,T} &= \rho_0\, u_{x_1} \\
m_5^{eq,T} &= \rho_0\, u_{x_2} \\
m_7^{eq,T} &= \rho_0\, u_{x_3} \\
m_9^{eq,T} &= 3p_{x_1x_1}^{eq,T} &&= \rho_0 \left(2\,u_{x_1}^2 - u_{x_2}^2 - u_{x_3}^2\right) \\
m_{11}^{eq,T} &= p_{x_3x_3}^{eq,T} &&= \rho_0 \left(u_{x_2}^2 - u_{x_3}^2\right) \\
m_{13}^{eq,T} &= p_{x_1x_2}^{eq,T} &&= \rho_0\, u_{x_1}\, u_{x_2} \\
m_{14}^{eq,T} &= p_{x_2x_3}^{eq,T} &&= \rho_0\, u_{x_1}\, u_{x_3} \\
m_{15}^{eq,T} &= p_{x_1x_3}^{eq,T} &&= \rho_0\, u_{x_1}\, u_{x_3}
\end{aligned} \tag{4.21}$$

Um den Spannungstensor zu erhalten und somit die in [51] erwähnten Inkonsistenzen im Interfacebereich für unterschiedliche Dichten zu vermeiden, müssen zudem folgende Korrekturterme bezüglich der Viskosität berücksichtigt werden:

$$\begin{aligned}
m_1^{eq,corr} &= 2\nu \left(u_{x_1}\partial_{x_1}\rho + u_{x_2}\partial_{x_2}\rho + u_{x_3}\partial_{x_3}\rho\right) \\
m_9^{eq,corr} &= 2\nu \left(2u_{x_1}\partial_{x_1}\rho - u_{x_2}\partial_{x_2}\rho - u_{x_3}\partial_{x_3}\rho\right) \\
m_{11}^{eq,corr} &= 2\nu \left(u_{x_2}\partial_{x_2}\rho - u_{x_3}\partial_{x_3}\rho\right) \\
m_{13}^{eq,corr} &= \nu \left(u_{x_1}\partial_{x_2}\rho + u_{x_2}\partial_{x_1}\rho\right) \\
m_{14}^{eq,corr} &= \nu \left(u_{x_2}\partial_{x_3}\rho + u_{x_3}\partial_{x_2}\rho\right) \\
m_{15}^{eq,corr} &= \nu \left(u_{x_1}\partial_{x_2}\rho + u_{x_3}\partial_{x_1}\rho\right)
\end{aligned} \tag{4.22}$$

Die Bestimmung der Ableitungen des Druckfeldes in Gl. 4.22 erfolgt mittels Finite-Differenzen.

Advektion des Phasenfeldes

Für die Advektion des Phasenfeldes ψ wird eine separate Lattice-Boltzmann-Gleichung gelöst:

$$g_{i,l}(t+\Delta t_l, \mathbf{x}+\mathbf{e_i}\Delta t_l) = g_{i,l}(t,\mathbf{x}) - \frac{\Delta t_l}{\tau_l}\left(g_{i,l}(t,\mathbf{x}) - g_i^{eq}\left(\psi(t,\mathbf{x}), \mathbf{u}(t,\mathbf{x})\right)\right) \tag{4.23}$$

Setzt man $\frac{\Delta t_l}{\tau_l} = 1$, erhält man folgende vereinfachte Gleichung:

$$g_{i,l}(t+\Delta t_l, \mathbf{x}+\mathbf{e_i}\Delta t_l) = g_i^{eq}\left(\psi(t,\mathbf{x}), \mathbf{u}(t,\mathbf{x})\right) \tag{4.24}$$

Die Gleichgewichtsverteilung ergibt sich zu:

$$g_i^{eq}(\psi, \mathbf{u}) = w_i \psi \left(1 + \frac{3}{c^2}\mathbf{e}_i \cdot \mathbf{u}\right) \tag{4.25}$$

Verwendet man Gl. 4.24 in Kombination mit Gl. 4.25, so erhält man eine Advektions-Diffusions-Gleichung [90]. Durch $\frac{\Delta t_l}{\tau_l} = 1$ erhält man einen Diffusionskoeffizienten $\alpha = \frac{1}{6}c^2\Delta t$, der zudem vom Gitterlevel abhängt. Um diesen Effekt zu beseitigen und die Vermischung der Phasen zu verhindern wird gemäß [154] eine Entmischung durchgeführt. Hierbei wird die Masse so umverteilt, dass das innere Produkt des Phasenfeldgradienten $\mathbf{C}_l$ und des Impulses der jeweiligen Phase $j^p = \sum_i \mathbf{e}_i f_i^p$ maximiert wird (Abb. 4.5). Hierbei muss gewährleistet sein, dass die Masse pro Phase sowie die Summe des Impulses beider Phasen lokal erhalten bleibt.

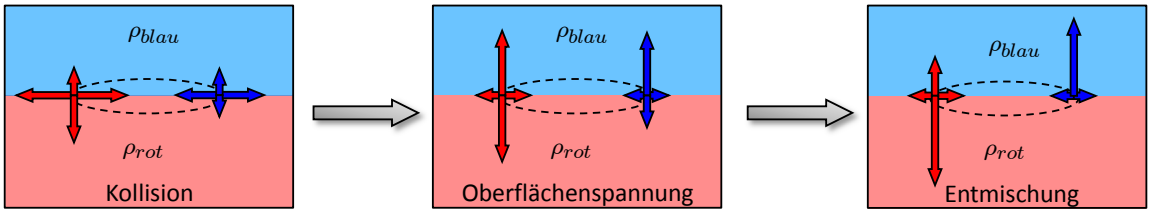


Abbildung 4.5: *Entmischung*

Zur Laufzeit wird an jedem Gitterknoten anhand seines Phasenfeldgradienten $\mathbf{C}_l$ entschieden, welche Algorithmen im Einzelnen durchgeführt werden müssen:

$|\mathbf{C}_l| > 0$:

Der Knoten befindet sich im Phaseninterfacebereich. Deshalb müssen bei der Relaxation des Strömungsfeldes die Oberflächenspannungsterme (Gl. 4.16 bzw. 4.17) berücksichtigt werden. Für die Entmischung werden anschließend die phasenspezifischen Verteilungen $g_{i,r}$ und $g_{i,b}$ gemäß Gl. 4.25 rekonstruiert:

$$g_{i,r}(t,\mathbf{x}) = w_i \psi_r(t,\mathbf{x})\left(1 + \frac{3}{c^2}\mathbf{e}_i \cdot \mathbf{u}(t,\mathbf{x})\right) \quad \text{mit} \quad \psi_r(t,\mathbf{x}) = \frac{1+\phi}{2}\psi(t,\mathbf{x}) \tag{4.26}$$

$$g_{i,b}(t,\mathbf{x}) = w_i \psi_b(t,\mathbf{x})\left(1 + \frac{3}{c^2}\mathbf{e}_i \cdot \mathbf{u}(t,\mathbf{x})\right) \quad \text{mit} \quad \psi_b(t,\mathbf{x}) = \frac{1-\phi}{2}\psi(t,\mathbf{x}) \tag{4.27}$$

Der Algorithmus für die Entmischung beim $d3q19$-Modell, in der die Stärke und somit der Gradient sowie die Ausdehnung des Phaseninterfaces mit Hilfe des Entmischungsfaktors $e \in]0,\ldots,1]$ gesteuert wird, lautet :

Algorithmus 1 Entmischung für das $d3q19$-Modell

for $i = 1$ **to** $i = 17$ **with** $\Delta i = 2$ **do**
  tmp $= e \cdot$ **min(** $g_{i,r}$, $g_{inv,r}$, $g_{i,b}$, $g_{inv,b}$ **)** $\mathbf{e}_i\mathbf{C}_l\,(|\mathbf{e}_i||\mathbf{C}_l|)^{-1}$
  $g_{i,r}$ $= g_{i,r}$ + tmp
  $g_{inv,r}$ $= g_{inv,r}$ − tmp
  $g_{i,b}$ $= g_{i,b}$ − tmp
  $g_{inv,b}$ $= g_{inv,b}$ + tmp
end for

Die anteilige Phasenfeldänderung der Nachbarknoten $\phi(t+\Delta t_l, \mathbf{x}) = \frac{\phi_Z(t+\Delta t_l,\mathbf{x})}{\phi_N(t+\Delta t_l,\mathbf{x})}$ wird anschließend wie folgt bestimmt:

$$\Delta\phi_{Z,i}(t+\Delta t_l,\mathbf{x}) = g_{i,r}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l) - g_{i,b}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l) \tag{4.28}$$

$$\Delta\phi_{N,i}(t+\Delta t_l,\mathbf{x}) = g_{i,r}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l) + g_{i,b}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l) \tag{4.29}$$

$|\mathbf{C}_l| = 0$:
Der Knoten befindet sich außerhalb des Phasenübergangbereichs. Die Berechnung der Oberflächenspannungsanteile als auch die Entmischung können somit vernachlässigt werden und die Advektion folgendermaßen vereinfacht werden:

$$\Delta\phi_{Z,i}(t+\Delta t_l,\mathbf{x}) = \phi(t,\mathbf{x}+\mathbf{e}_i\Delta t)\; g_i^{eq}\left(\psi(t,\mathbf{x}+\mathbf{e}_i\Delta t_l),\mathbf{u}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l)\right) \tag{4.30}$$

$$\Delta\phi_{N,i}(t+\Delta t_l,\mathbf{x}) = g_i^{eq}\left(\psi(t,\mathbf{x}+\mathbf{e}_i\Delta t),\mathbf{u}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l)\right) \tag{4.31}$$

Nach jedem Zeitschritt ergibt sich ϕ und die neue Phasenfelddichte ψ zu:

$$\phi(t+\Delta t_l,\mathbf{x}) = \frac{\phi_Z(t+\Delta t_l,\mathbf{x})}{\phi_N(t+\Delta t_l,\mathbf{x})} = \frac{\sum_i \Delta\phi_{Z,i}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l)}{\sum_i \Delta\phi_{N,i}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l)} \tag{4.32}$$

$$\psi(t+\Delta t_l,\mathbf{x}) = \sum_i \Delta\phi_{N,i}(t,\mathbf{x}+\mathbf{e}_i\Delta t_l) \tag{4.33}$$

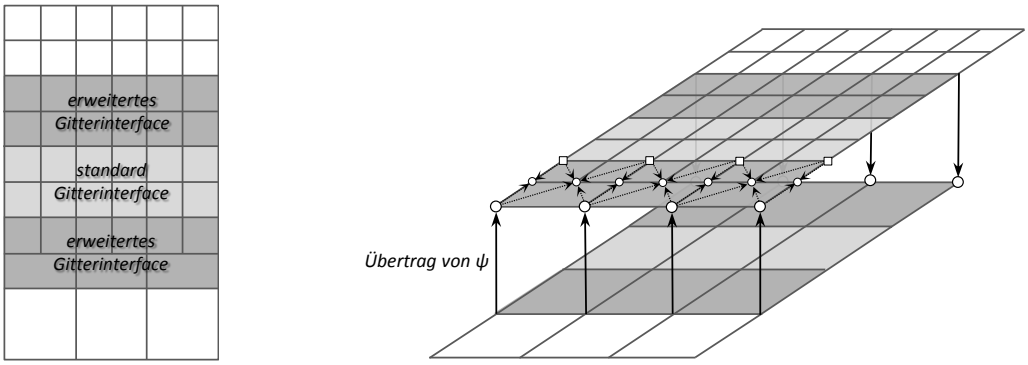
Eine mathematische Analyse zeigt, dass dieses numerische Verfahren die Navier-Stokes-Gleichungen für zwei nicht-mischbare Phasen unter Einbezug der Oberflächenspannung nachbildet [90].

Bei der Implementierung hat die getrennte Betrachtung des Strömungs- bzw. Phasenfeldes den Vorteil, dass pro Gitterknoten insgesamt nur ein Verteilungssatz vorgehalten werden muss (zwei wenn man zur Propagation ein separates Feld verwendet). Zur Bestimmung von $\mathbf{C}_l$ und g_i werden zusätzlich Variablen für die Phasenfelddichte ψ und ϕ_N sowie zwei weitere zur Advektion des Phasenfeldes (ϕ_Z und ϕ_N) benötigt. Im Ursprungsmodell wurde für jede Phase ein Verteilungssatz benötigt.

4.3.2 Nicht-uniforme Gitter

Im Vergleich zu Berechnungen für Einphasenströmungen auf nicht-uniformen Gittern müssen zur Bestimmung des Phasenfeldgradienten $\mathbf{C}_l$ am Gitterübergang zusätzlich die fehlenden Phasenfeldinformationen der groben bzw. feinen Nachbarknoten bestimmt werden. Da es sich hierbei um eine makroskopische Größe handelt, ist diese gitterlevelunabhängig und kann ohne Skalierung aus dem anderen Level bezogen werden. Auch die Advektion des Phasenfeldes über das Gitterinterface hinweg bedarf keiner Skalierung, da mit Gl. 4.25 ausschließlich levelunabhängige Gleichgewichtsanteile propagiert werden.

Im feinen Level sind jedoch zusätzliche Raum- und Zeitinterpolationen nötig (Abb. 4.6).



(a) *Erweitertes Gitterinterface* ***(b)*** *Übertrag und Rauminterpolation von* ψ

Abbildung 4.6: *Gitterinterface für Mehrphasenprobleme*

Es wurden verschiedene algorithmische Varianten und deren Kombinationen implementiert und getestet:

- diverse Diskretisierungssterne am Gitterinterface zur Bestimmung des Phasenfeldgradienten
- konstante, lineare und quadratische Zeitinterpolation der Phasenfelddichte zur Berechnung des Phasenfeldgradienten
- Vernachlässigung der Oberflächenspannung
- Skalierung aller bzw. lediglich der benötigten Verteilungen an einem Knoten im Überlappungsbereich

Die besten Ergebnisse wurden für den Fall erzielt, bei dem lediglich die fehlenden Verteilungen eines Knotens gemäß Abschn. 3.3 skaliert und interpoliert wurden. Die Berechnung des Phasenfeldgradienten erfolgte dabei nach Gl. 4.12 im Anschluss an die in Abb. 4.6b dargestellte räumliche Interpolation der zeitlich linear interpolierten Phasenfeldwerte. Dennoch traten beim Aufeinandertreffen von Phasen- und Gitterinterface weiterhin Effekte auf, die je nach der Wahl der Strömungsparameter unterschiedlich stark ausfielen. Ähnlich wie bei der Simulation von Einphasenproblemen auf nichtuniformen Gittern führte die Verwendung von Raum- und Zeitinterpolationen mit Ansätzen höherer Ordnung zu keiner nennenswerten Verbesserung.

4.3.2.1 Beispiel: Advektion einer Blase auf einem a priori verfeinerten, statischen Gitter

Anhand dieses Beispiels sollen qualitativ die Probleme aufgezeigt werden, die beim Aufeinandertreffen des physikalischen Phaseninterfaces auf das Gitterinterface auftreten. Hierzu wird eine Blase im uniformen 2-D-Strömungsgebiet initialisiert und über einen lokal verfeinerten Bereich advektiert. Das Gebiet ist in beide Richtungen periodisch. Wie in Abb. 4.7 zu erkennen ist, unterscheidet sich die Ausdehnung des Phaseninterfaces in den unterschiedlichen Gitterleveln. Die Breite beträgt immer $d \cdot \Delta x_l$. Der konstante Faktor d hängt vom gewählten Entmischungsalgorithmus ab und ist anders als Δx_l gitterlevelunabhängig (Abb. 4.8). Integriert man die Differenz der normalen und der tangentialen Komponente des Drucktensors, so erhält man die gleiche, korrekte Oberflächenspannung für jeden Level.

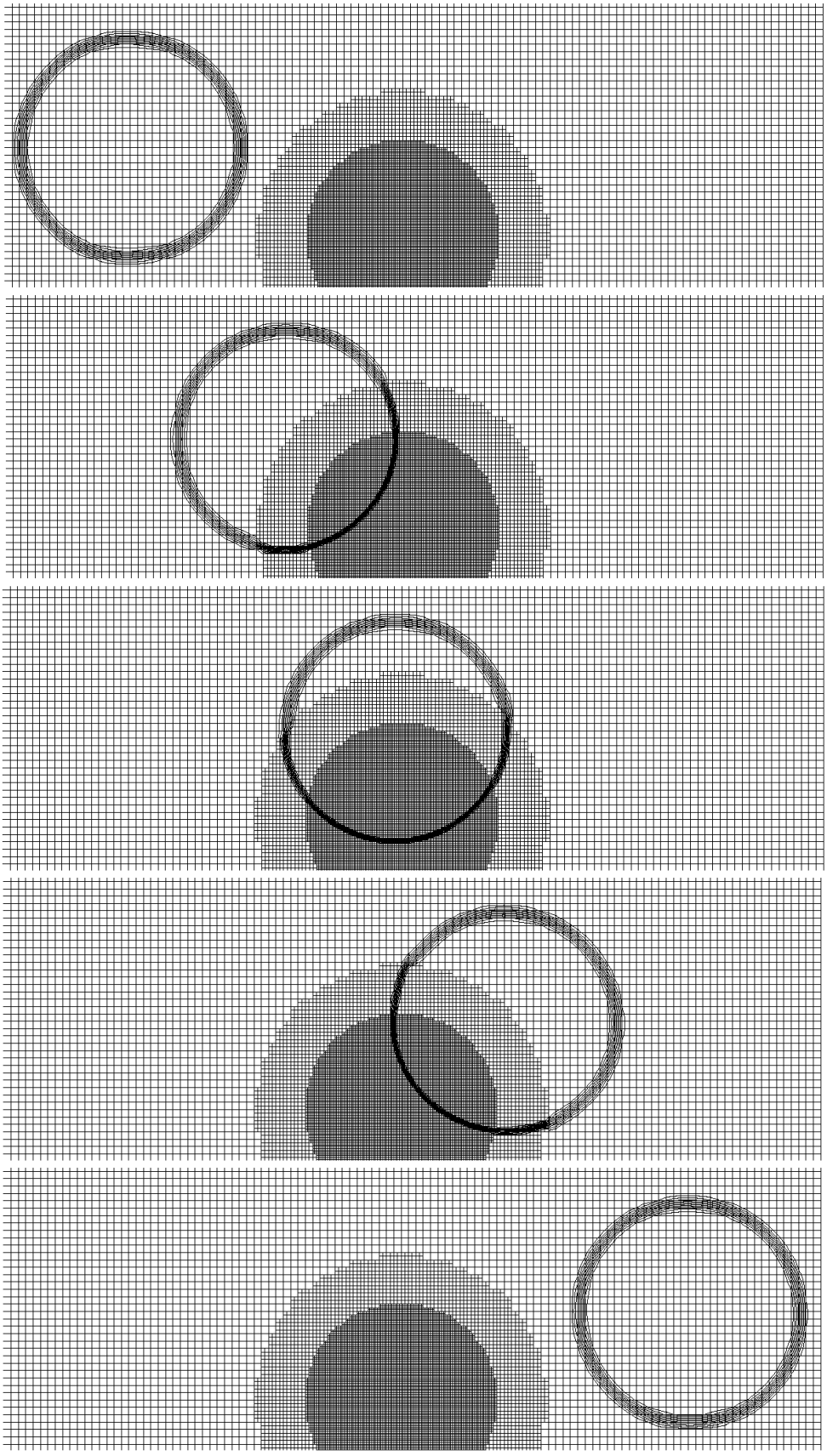


Abbildung 4.7: *Isolinien für ϕ einer advektiv transportierten Blase auf einem Baumgitter ($\sigma = 0,01$ N/m)*

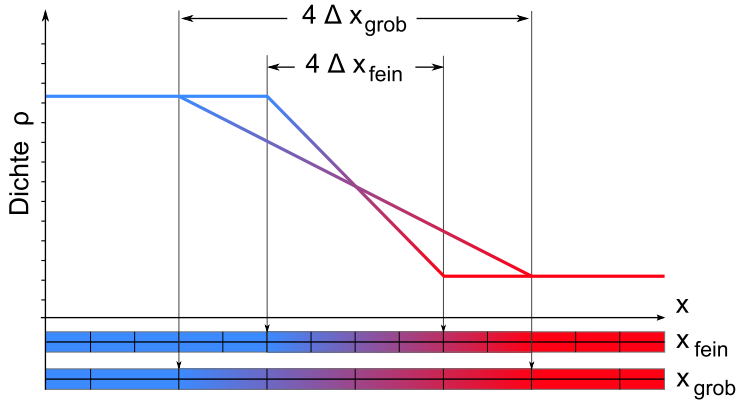
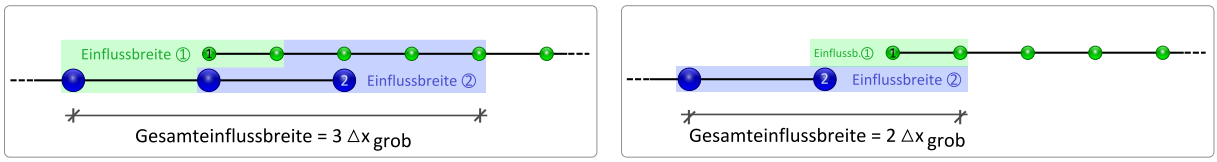


Abbildung 4.8: *Idealisierter Dichteverlauf am Phaseninterface für zwei Gitterlevel*

Abhängig von der Größe der Oberflächenspannung treten Störungen im System auf. Vernachlässigt man die Oberflächenspannung, so erhält man ein reines Advektionsproblem, und die Oberflächenform der transportierten Blase wird geringfügig deformiert (Abb. 4.9a). Für große Oberflächenspannungen hingegen werden störende künstliche Kapillarkräfte induziert (Abb. 4.9c).

Um störende Einflüsse durch den großen Einflussbereich des Gitterüberganges zu minimieren, wurde für diesen ein neuer Kopplungsalgorithmus entwickelt. Insbesondere in Bezug auf die Bestimmung des Phasenfeldgradienten wurde die Einflussbreite deutlich verringert (Abb. 4.10). Zudem befindet sich im Knotengitter an jedem Ort **x** nur noch maximal ein Knoten, wodurch redundante Informationen vermieden wurden. Damit konnte zwar die Ausdehnung der Störung reduziert werden, die Amplitude der Störung erhöhte sich jedoch.



(a) *Alter Kopplungsalgortihmus* ***(b)*** *Neuer Kopplungsalgortihmus*

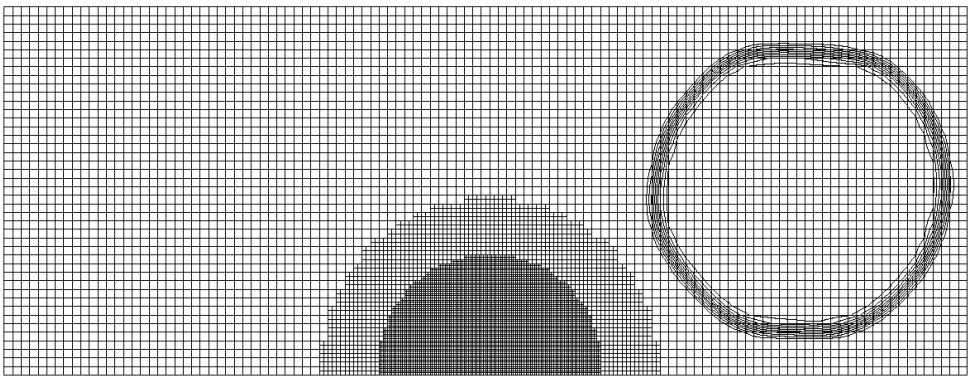
Abbildung 4.10: *Vergleich der Einflussbreiten*

4.3.2.2 Stabilität: Bestimmung der maximalen Anzahl an Gitterlevel

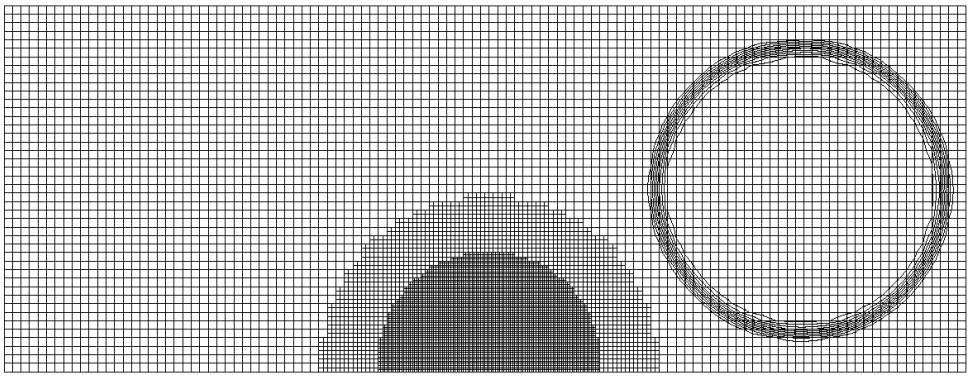
Die maximale Anzahl von Verfeinerungsstufen in einem System hängt vom Kollisionsfaktor ω ab. Dieser ist maßgebend für die Stabilität des LB-Verfahrens und ergibt sich für den Gitterlevel l in Bezug auf den Basislevel L und der kinematischen Viskosität ν zu:

$$\omega_l = \frac{\Delta t_l}{\tau_l} \quad \text{mit} \quad \tau = 3\frac{\nu}{c^2} + \frac{1}{2}\Delta t_l \quad \text{und} \quad \Delta t_l = \frac{\Delta t_L}{2^{l-L}}. \tag{4.34}$$

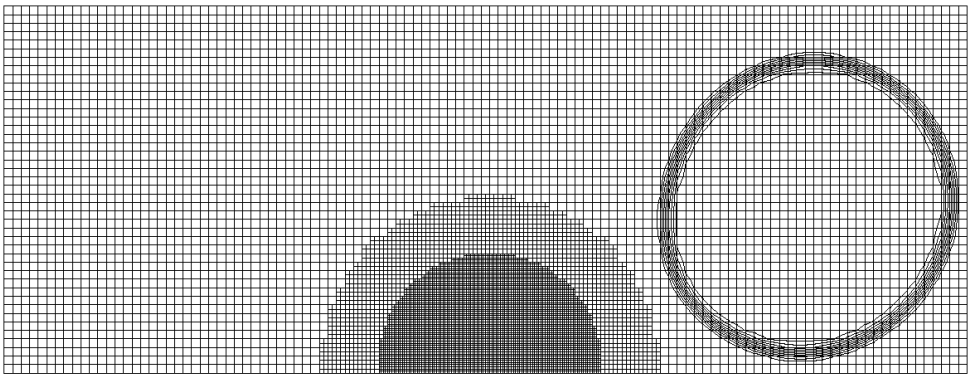
Die Gültigkeitsgrenzen von ω sind vom konkreten Strömungsproblem und verwendeten -modell abhängig, können jedoch in erster Näherung abgeschätzt werden. Die untere Schranke ω_{min} wird durch die Einschränkung definiert, dass das LB-Verfahren nur für eine kleine Knudsenzahl Gültigkeit bezüglich der NS-Gleichungen besitzt. Die generelle Stabilität gibt den oberen Schwellwert ω_{max} vor. Im Allgemeinen verwendet man bei Berechnungen mit dem BGK-Modell $\omega_{min} = \frac{2}{3}$ und $\omega_{max} \approx 1,976$, und mit der MRT-Annäherung $\omega_{min} = \frac{2}{3}$ und $\omega_{max} \approx 1,999$. Für Mehrphasenprobleme mit großen



(a) $\sigma = 0,0\ N/m$



(b) $\sigma = 0,001\ N/m$



(c) $\sigma = 0,01\ N/m$

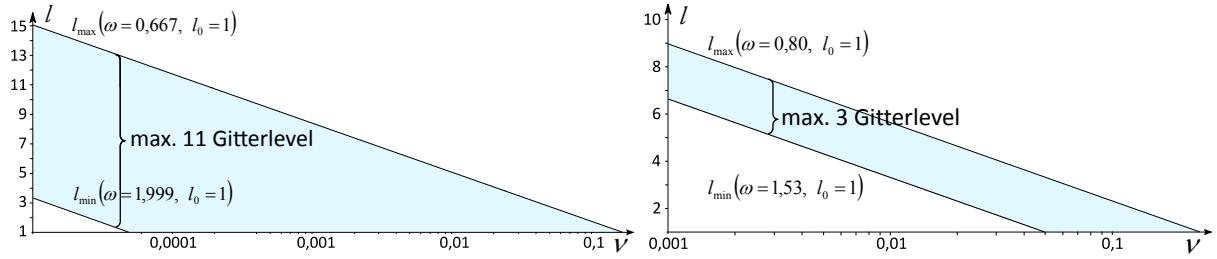
Abbildung 4.9: *Isolinien für ϕ einer advektiv transportierten Blase auf einem Baumgitter*

Kapillarkräften als Ergebnis hoher Oberflächenspannung muss sich der Kollisionsfaktor im Bereich $0,8 < \omega < 1,5$ befinden. Aus Gl. 4.34 erhält man [22]:

$$\omega_l = \frac{\Delta t_l}{\tau_l} = \frac{\left(\frac{1}{2}\right)^{l-L}}{3\frac{\nu}{\Delta t_L c^2} + \left(\frac{1}{2}\right)^{l-L+1}} \tag{4.35}$$

Der zugehörige Gitterlevel l lässt sich demzufolge folgendermaßen bestimmen:

$$l = L - \log_2 \frac{6\nu\omega_l}{\Delta t_L c^2 (2-\omega_l)} \tag{4.36}$$



***(a)** Einphasenprobleme (MRT)* ***(b)** Mehrphasenprobleme mit hohen Kapilarkräften*

***Abbildung 4.11:** Maximale Anzahl unterschiedlicher Gitterlevel in Abhängigkeit der kinematischen Viskosität*

Setzt man für die unterschiedlichen Modelle die jeweiligen Grenzen von ω ein, so ergibt sich die in Abb. 4.11a und b dargestelle Höchstanzahl an Gitterleveln in Abhängigkeit der Viskosität. Während für Einphasenprobleme unter Verwendung des MRT-Modells bis zu elf Level verwendet werden können, sind es für Mehrphasenprobleme gerade einmal drei, wodurch die Flexibilität stark eingeschränkt wird. Eine Übersicht über die mögliche Levelanzahl gibt Tab. 4.1.

Modell	ω_{min}	ω_{max}	#Gitterlevel
Einphase (BGK)	$\approx 0{,}67$	$\approx 1{,}976$	≈ 7
Einphase (MRT)	$\approx 0{,}67$	$\approx 1{,}999$	≈ 11
Mehrphase (MRT)	$\approx 0{,}80$	$\approx 1{,}500$	≈ 3

***Tabelle 4.1:** Modellabhängige, maximale Anzahl von Gitterleveln*

5 Gittertypen

Die in dieser Arbeit beschriebenen Lattice-Boltzmann-Modelle basieren auf regulären Gittern. Ist man nicht an einer partiellen Gebietsverfeinerung interessiert, genügt eine matrixartige Datenstruktur zur Repräsentation des Berechnungsgitters. Bei Verwendung von nicht-uniformen Gittern bieten sich Datenstrukturen an, die auf einer Quad- bzw. Octreedatenstruktur aufsetzen. In dieser Arbeit wird zwischen drei verschiedenen Gittertypen unterschieden: uniformes, a priori verfeinertes und adaptives Gitter. Wann genau welches Gitter eingesetzt werden sollte, hängt von dem jeweiligen Problem und verwendeten Modell ab. So kann das Einphasenmodell auf jedem Gittertyp angewendet werden, beim erweiterten RK-Modell hingegen können bei Verwendung a priori verfeinerter Gitter und bestimmter Parameter Störungen auftreten, sobald das physikalische Phaseninterface auf das geometrische Gitterinterface trifft. Hinzu kommt, dass die maximale Anzahl an möglicher Gitterlevel bei Mehrphasensimulationen sehr stark eingeschränkt ist (vgl. Abs. 4.3.2). Generell ist dort eine Struktur vorzuziehen, die die Phasenübergangsbereiche kontinuierlich mit der höchsten Verfeinerungsstufe auflöst. Diese Anforderung kann nur durch eine dynamische Gitterstruktur erfüllt werden, bei der sich das Rechengitter zur Laufzeit stetig an das Strömungsproblem anpasst. Dies stellt eine große Herausforderung für die zu entwickelnde Datenstruktur dar.

Abb. 5.1 gibt einen Überblick der verschiedenen Berechnungsgitter und vergleicht deren Anwendbarkeit in Bezug auf deren Effizienz für die hier beschriebenen Ein- und Mehrphasenmodelle.

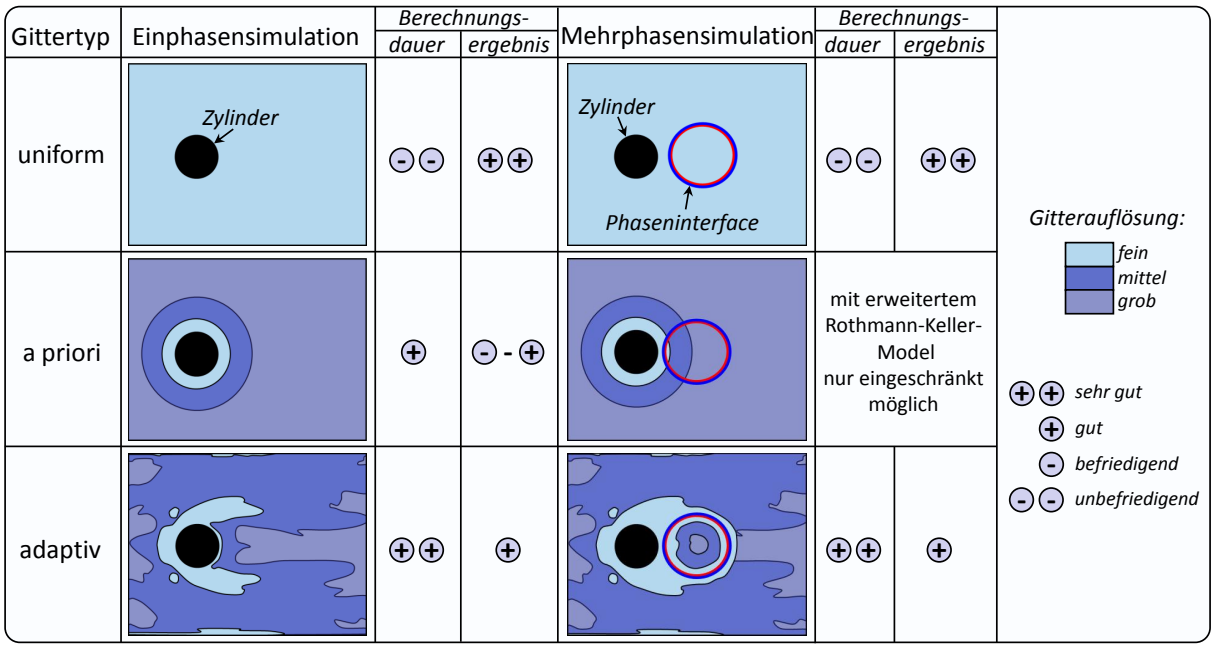


Abbildung 5.1: *Vergleich von verschiedenen Berechnungsgittern*

Teil II

VIRTUALFLUIDS auf Knotengittern

Überblick

In folgenden Teil der Arbeit wird ein Softwarekonzept für einen Lattice-Boltzmann-Strömungssimulator auf Basis baumhierarchischer Knotengitter vorgestellt. Beim Entwurf wurde besonderes Augenmerk auf einen hohen Abstrahierungsgrad und die Wiederverwendbarkeit des Programmcodes gelegt. Ziel war ein modulares Softwarepaket, das mit minimalem Aufwand hinsichtlich weiterer Lattice-Boltzmann-Berechnungskerne ergänzt werden kann. Ein besonderes Merkmal ist hierbei, dass Fluidknoten durch Knotenobjekte repräsentiert werden, die in quad- bzw. octreeartigen Datenstrukturen im Rechner vorgehalten werden.

Im Folgenden werden zunächst die zum Verständnis wesentlichen Grundlagen des Softwarepakets, wie z. B. dessen Aufbau und Datenstruktur, erläutert. Anschließend wird die Erweiterung hinsichtlich einer adaptiven Gittersteuerug besprochen und anhand einer aufsteigenden Blase veranschaulicht. Vervollständigt werden diese Grundlagen durch serielle Validierungen des 3-D-Kerns für Ein- und Zweiphasenprobleme, für die unter anderem der Vorteil der adaptiven Gitter aufgezeigt wird.

Der letzte Teil dieses Kapitels beschäftigt sich mit der Erweiterung des Softwarepakets hinsichtlich verteilter Berechnungskerne für den 2-D-Löser. Es wird unter anderem ein abstraktes, benutzerdefinierbares Regelwerk vorgestellt, mit dessen Hilfe die Gebietszerlegung durchgeführt wird. Neben Messungen der parallelen Effizienz und der Skalierbarkeit des Lösers werden zur Validierung die Simulationsergebnisse einer Tandemzylinderumströmung vorgestellt.

6 Basiskonzept

Der während dieser Arbeit entwickelte Lattice-Boltzmann-Strömungslöser VIRTUALFLUIDS wurde für 2-D- und 3-D-Probleme entwickelt. Eines der Hauptziele war es sowohl einen effizienten als auch flexiblen und vor allem jederzeit erweiterbaren numerischen Strömungslöser zu generieren. Aus diesem Grund fiel beim Entwurf der Software die Entscheidung für einen objektorientierten Ansatz unter Berücksichtigung der in [14, 38, 43] beschriebenen Entwurfsmuster. Zur Wahrung der Flexibilität, wurde C++ als Programmiersprache gewählt. Die ursprünglich in den 1980er Jahren von Bjarne Stroustrup entwickelte, sehr verbreitete Sprache unterstützt neben der Objektorientierung sowohl das generische als auch das durch Fortran und C bekannte prozedurale Programmierparadigma. Dies ermöglichte eine effiziente und maschinennahe Programmierung.

6.1 Datenstruktur

Als zugrunde liegenden Datenstruktur kommt der in [11] beschriebene Quad- bzw. Octree zum Einsatz.

Ein *Quad-/Octree* ist aus der Sicht der Informatik ein gewurzelter Baum, dessen Knoten entweder genau vier/acht Nachfolger haben oder keinen. Knoten ohne Vorgänger werden hierbei als *Wurzel* (*engl.: root*) und Knoten ohne Nachfolger als *Blätter* (*engl.: leaves*) bezeichnet.

Da das hier verwendete LB-Verfahren auf einem Finite-Differenzen-Schema basiert, werden die Blätter der Baumstruktur nicht durch Zell- sondern durch Knotenobjekte repräsentiert.

Der Ansatz sieht vor, Topologie und physikalische Daten strikt zu trennen. Dadurch kann die Topologie für eine Vielzahl physikalischer Modelle verwendet werden. Übertragen auf die gestellten Anforderungen wurde ein entsprechendes Softwarekonzept entwickelt. Eine Übersicht über die zum Verständnis wesentlichen Klassen des 3-D-Lösers gibt Abb. 6.1. Das Softwarepaket gliedert sich dabei in drei unterschiedliche Schichten:

topology layer

Die Topologieschicht kapselt die Funktionalität, Octreegitter (OctNodeGrids) unabhängig von der Physik zu generieren, zu verändern und zu serialisieren. Auf eine Datenrepräsentation in Form einer für derartige Bäume häufig verwendeten verzeigerten Datenstruktur [22] wurde bewusst verzichtet. Um Ressourcen zu sparen, werden ausschließlich die Blätter (OctNodes) levelweise in assoziativen Arrays (OctNodeMap mit Hashtabelle als Containertyp) gespeichert. An einem Punkt **x** kann sich hierbei maximal ein OctNode befinden. Um die Navigation durch den Baum zu ermöglichen, werden in dessen Blättern anstelle der tatsächlichen Weltkoordinaten levelbezogene Indizes $\mathbf{i}_L$ (vgl. Abb. 6.2) vorgehalten:

$$\mathbf{i}_L = \left(\text{Index}_{x_1}, \text{Index}_{x_2}, \text{Index}_{x_3}\right)^T \tag{6.1}$$

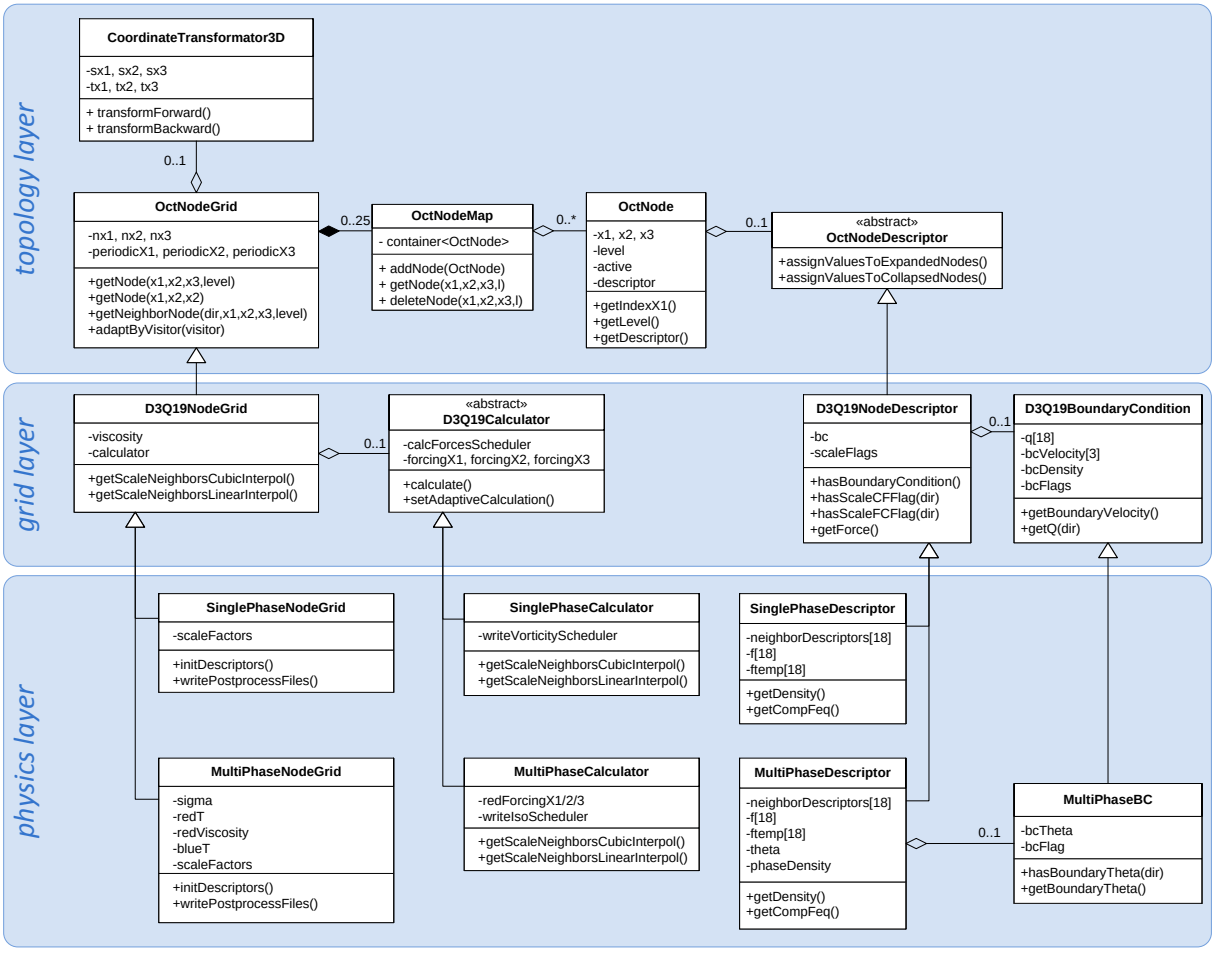


Abbildung 6.1: *Knotengitter (UML)*

Diese Indizes dienen als Basis für die Schlüsselgenerierung, die für die Adressierung in der OctNodeMap notwendig ist. Durch simple Addition und Subtraktion der entsprechenden Indizes können die Positionen von Nachbarknoten ermittelt werden und anschließend auf Vorhandensein in der OctNodeMap überprüft werden. Das Durchlaufen von Suchpfaden, wie bei einem klassischen Octree nötig, entfällt.

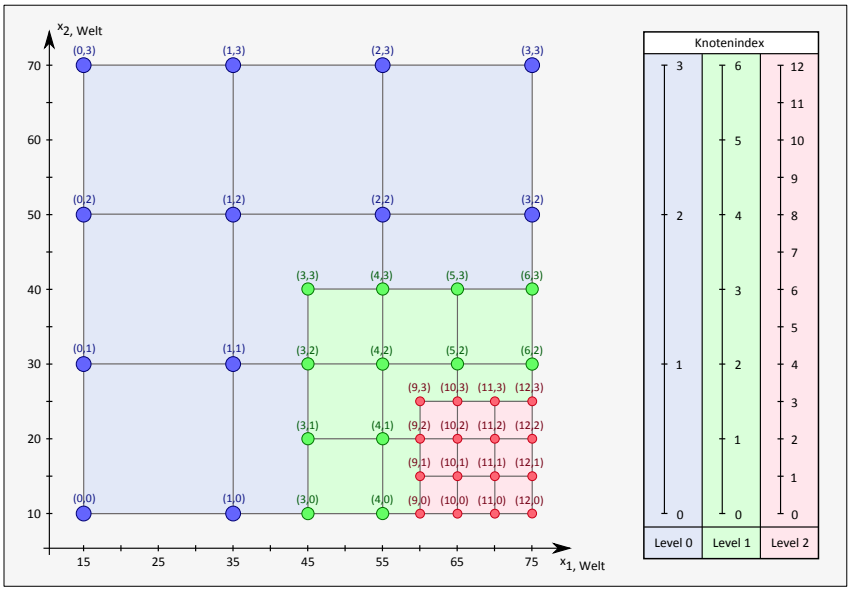


Abbildung 6.2: *Knotenindizierung für verschiedene Gitterlevel (2-D)*

Für die Navigation zwischen den einzelnen Gitterleveln L des Knotengitters gilt folgende Beziehung:

$$\mathbf{i}_{L_A} = 2^{\Delta L}\,\mathbf{i}_{L_B} \quad \text{mit} \quad \Delta L = L_A - L_B \tag{6.2}$$

In der Implementierung wird hierfür Ganzzahlarithmetik im Zusammenspiel mit Bitmanipulationen verwendet. So hat sich die Verwendung von Shift-Operatoren (« und ») an Stelle der pow-Funktion aus der Standard-Mathematik-Bibliothek von C++ als sehr effizient erwiesen. Die Transformation in das Weltkoordinatensystem, die durch das CoordinateTransformator3D-Modul durchgeführt wird, erfolgt vereinfacht unter Vernachlässigung von Verdrehungen:

$$\mathbf{x}_{L_A,Welt}\left(\mathbf{i}_{L_A}\right) = \mathbf{x}_{L_0,Welt}(\mathbf{0}) + \frac{\mathbf{i}_{L_A}}{2^{L_A}}\,\Delta\mathbf{x}_{L_0,Welt} \tag{6.3}$$

Die Weltkoordinaten des Indexursprungs $\mathbf{x}_{L_0,Welt}(\mathbf{0})$ sowie der Knotenabstand $\Delta\mathbf{x}_{L_0,Welt}$ für Level 0 können hier vom Benutzer festgelegt werden.

Um später Instanzen von OctNode (unterschiedliche) Eigenschaften zuweisen zu können, wird als Schnittstelle die Interfaceklasse OctNodeDescriptor zur Verfügung gestellt. Durch Verwendung dieses Interfaces können mit minimalem Aufwand ohne topologische Einschränkungen den Berechnungsgittern Knoten mit unterschiedlichen physikalischen Eigenschaften zugewiesen werden.

grid layer

Die Komponenten dieser Zwischenschicht vereinen unter anderem die gemeinsamen Attribute und Methoden der verschiedenen LB-Löser des *physics layers* und stellen diese generalisiert zur Verfügung.

Das spezielle, vom allgemeinen OctNodeGrid abgeleitete D3Q19NodeGrid definiert beispielsweise die für das $d3q19$-Modell charakteristischen Diskretisierungsrichtungen sowie einige Strömungsparameter. Darüber hinaus stellt es eine Vielzahl von Hilfsmethoden zur Verfügung. Hierzu gehört etwa die Ermittlung der zu einem Gitterinterfaceknoten zugehörigen Skalierungs- bzw. Interpolationsnachbarnoten.

Mit dem Modul D3Q19Calculator wird die Basisklasse eingeführt, deren Spezialisierungen für die LB-Simulationen verwendet werden. Auch diese enthält wiederum Parameter und Methoden, die von den abgeleiteten Klassen verwendet werden.

Der D3Q19NodeDescriptor speichert in erster Linie den boltzmannspezifischen Knotentyp, der den Knoten während des Präprozesses zugewiesen wird (Abb. 6.3). Dieser ist wichtig, da zum Beispiel Skalierungs- bzw. Randbedingungsknoten zur Berechnungslaufzeit gesondert behandelt werden müssen (vgl. Abschn. 3.2 und 3.3). Die Knoten werden hierzu mit Hilfe von Bitmasken mit richtungsbezogenen Knotenflags versehen. Während die Skalierungsinformation direkt im D3Q19NodeDescriptor vorgehalten wird, wurden die Randbedingungsflags zusammen mit den jeweils zugehörigen makroskopischen Werten zur Minimierung des Speicherbedarfs in die D3Q19BoundaryCondition ausgelagert. In dieser werden auch für Knoten, die sich z. B. in der Nähe von Randbedingungsobjekten, wie Festkörpern, befinden, die normierten ebenfalls richtungsabhängigen Abstände **q** gespeichert.

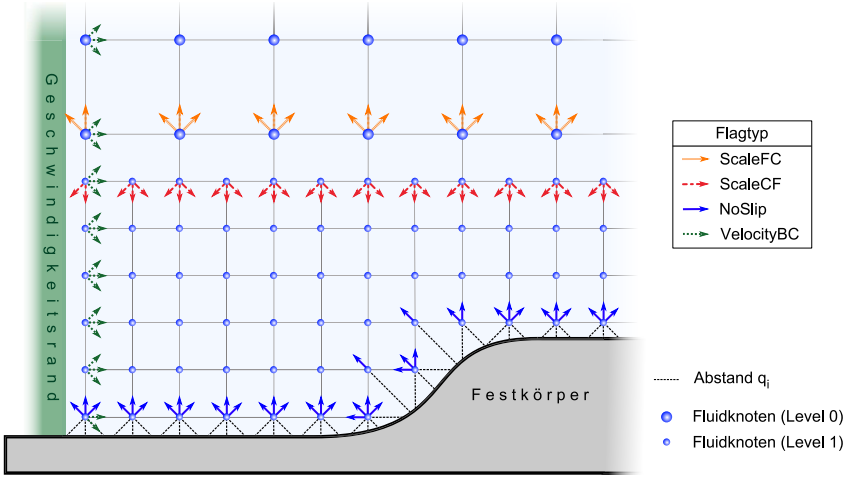


Abbildung 6.3: *Knotenflags*

Während q_i dabei immer in die Richtung des Randes zeigt, wird das Randbedingungsflag in die entgegengesetzte Richtung gesetzt. Dies entspricht der Richtung, in der der Knoten aufgrund des Randes in der Propagation keine gültige Verteilung erhält.

physics layer

Hier befinden sich die eigentlichen Berechnungskerne für verschiedene physikalische Modelle (Einphase, Mehrphase, etc.). Die jeweiligen Gitter werden um spezielle Attribute, wie Viskosität, Skalierungsfaktoren und Oberflächenspannung, erweitert. Die Berechnungsklasse implementiert die Zeitschleife, definiert das Verhalten bei Kollision, Propagation, Skalierung und realisiert die Randbedingungen.

Die einzelnen Deskriptoren besitzen neben den Verteilungen und anderen Knotenwerten oftmals optimierte Berechnungsvorschriften für die makroskopischen Werte. So wird bei Mehrphasenberech-

nungen die Dichte oder Geschwindigkeit nur dann neu berechnet, wenn sich eine Verteilung geändert hat. Für eine effiziente Simulation werden den einzelnen Deskriptoren für einen direkten Zugriff die jeweiligen Nachbardeskriptoren zugewiesen. Der Deskriptor ist somit einer der wichtigsten Komponenten des Konzepts, der insbesondere durch seine Eigenständigkeit maßgeblich zur Flexibilität beiträgt.

Anmerkungen

Weitere Informationen und Erweiterungen des hier vorgestellten Ansatzes, wie Gittervisualisierung und interaktive Steuerung unter Verwendung der Bibliotheken *Qt* [111] und *VTK* [91], finden sich in der Dissertation von Sebastian Geller [46], der dieses Konzept mitentwickelt und -umgesetzt hat. Im Laufe der Zeit wurde das Knotengitterpaket um diverse Löser erweitert. Hierzu gehören Berechnungskerne für Sedimenttransport [144], freie Oberflächen [83, 95, 157] und Fluid-Struktur-Interaktion [48]. In dieser Arbeit wird jedoch nur auf die Implementierung des Ein- bzw. Mehrphasenkerns eingegangen.

6.2 Verbesserung der Datenlokalität

Die Verwendung von Hashtabellen als Speicherstruktur für die Quad- und Octreegitter ermöglicht einen schnellen Zugriff auf den jeweiligen Knoten. Die verwendete Hashtabelle besteht aus einem Datenvektor, der als Datenelemente Listen mit Knoten vorhält. Mittels einer Hashfunktion $h(k)$ wird anhand der Knotenpositionindizes ein Schlüssel generiert, der dem zugehörigen Vektorindex entspricht. Bei dem hier verwendeten geschlossenem Hashing können dadurch zwei verschiedene Knoten den gleichen Schlüssel aufweisen. In einem solchen Fall spricht man von einer Kollision und beide Knoten werden in der Liste an der zugehörigen Vektorposition angehängt. Um die Anzahl an Kollisionen zu minimieren, muss die Hashfunktion die Schlüssel möglichst gleichmäßig über den Indexbereich des Vektors verteilen. Dies steht jedoch im Widerspruch zu der gewünschten Datenlokalität, da benachbarte Knoten auch datenlokal im Speicher abgelegt werden sollten, um zusätzliche Speicherzugriffe zur Laufzeit zu vermeiden. Bei einer optimalen Hashfunktion wäre dies nicht der Fall. Da die Berechnungen später ausschließlich auf Vektoren mit den spezifischen Knotendeskriptoren stattfinden, ist eine Optimierung hinsichtlich deren Datenlokalität ausreichend. Diese Aufgabe übernimmt der ChessMemPool2D/3D, der je nach Bedarf de- bzw. aktiviert werden kann.

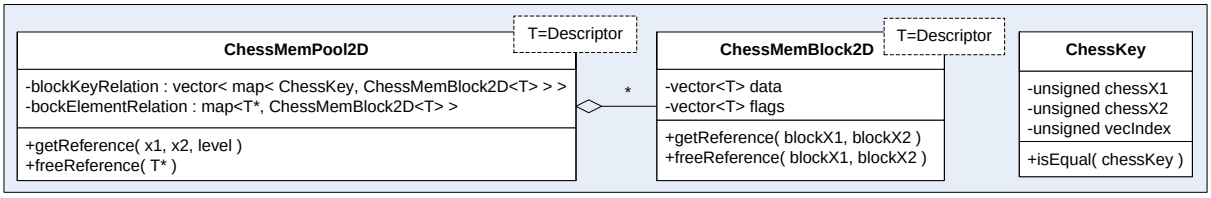


***Abbildung 6.4:** ChessMemPool (UML)*

Der ChessMemPool verwaltet levelweise quadratische/kubische ChessMemBlock2D/3D-Objekte, die in VIRTUALFLUIDS wiederum eine zum Programmstart definierte Anzahl an Deskriptoren verwalten (Abb. 6.5). Mittels Überladen des `new`- und des `delete`-Operators der spezifischen Deskriptorklasse werden Referenzen auf den entsprechenden Speicherbereich innerhalb des ChessMemBlocks erstellt bzw. gelöscht. Hierzu ermittelt der ChessMemPool aus den Knotenindizes den entsprechenden `ChessKey` und initialisiert, wenn noch nicht vorhanden, den zugehörigen ChessMemBlock.

Auch wenn nur **ein** Element innerhalb eines neuen ChessMemBlocks angefordert wird, ist dessen vollständige Initialisierung notwendig. Das resultiert in einem minimal höheren Speicheraufwand, kann aber durch die im Schnitt hinzugewonnene Leistungssteigerung von 20 % bei minimal eingeschränkter Flexibilität vernachlässigt werden. Die Datenhaltung innerhalb eines ChessMemBlocks erfolgt mit Hilfe eines Füllvektors, der entsprechend Abb. 6.5 den Block durchläuft. Diesem Vektor entsprechend werden die Deskriptoren in Datenvektoren gespeichert. Ein weiterer Vektor hält die Information vor, ob und auf welche Elemente referenziert wird. Wird auf kein Objekt referenziert, wird der ChessMemBlock automatisch deallokiert.

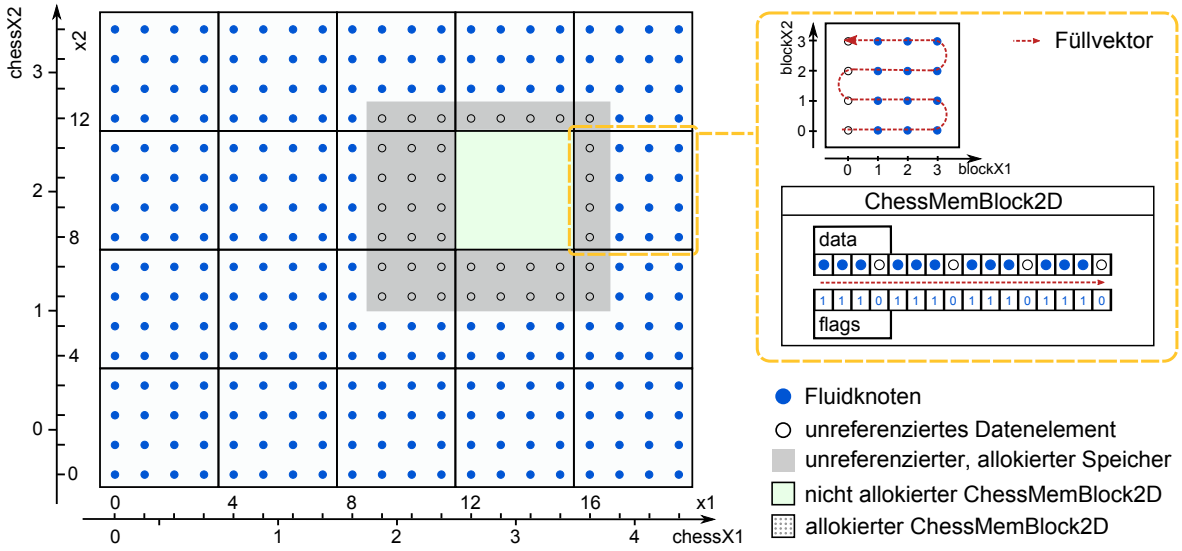


Abbildung 6.5: *Blockeinteilung beim ChessMemPool2D für einen Gitterlevel*

Eine weitere Möglichkeit die Datenstruktur zu optimieren ist in [39, 93, 136] zu finden. Bei diesem Ansatz werden die einzelnen Verteilungen f optimiert in Vektoren gehalten und ggf. nach bestimmten Kriterien (Fluid, Druckrand, Geschwindigkeitsrand, etc.) sortiert.

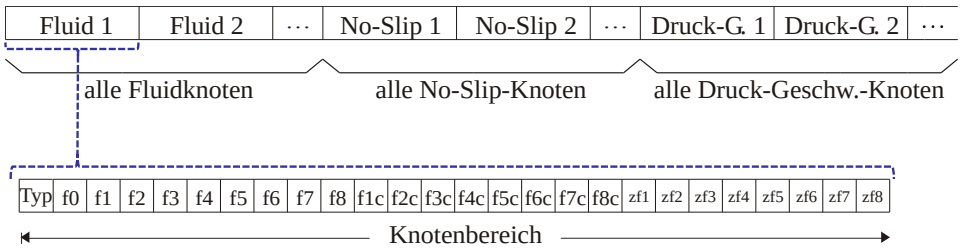
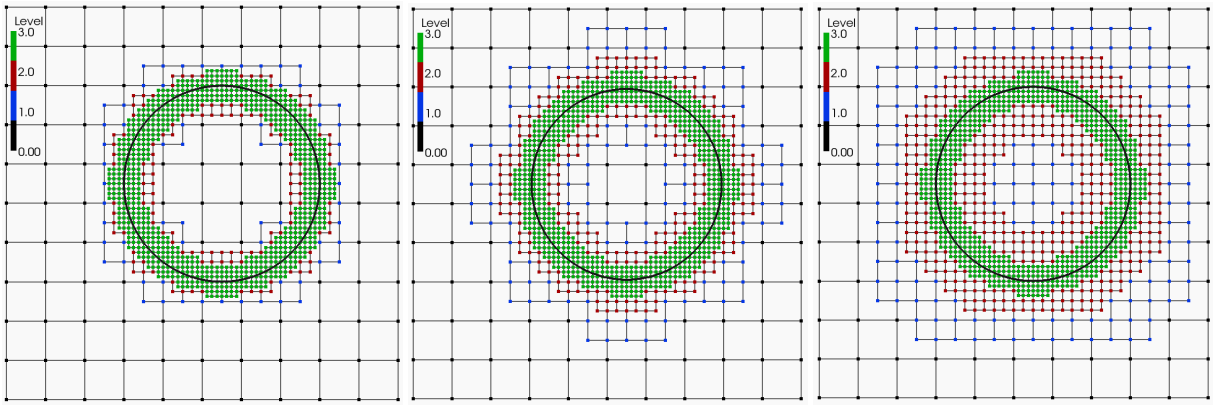


Abbildung 6.6: *Beispiel für Abbildung des Berechnungsgitters auf eine reine Vektordatenstruktur*

Alle LB-Algorithmen finden anschließend auf diesen Datenvektoren statt. Allerdings ist durch diese Optimierung die Flexibilität anschließend stark eingeschränkt und eine Erweiterung wenn überhaupt nur mit hohem Aufwand möglich und bietet sich vor allem für Löser mit uniformen Gitter an. In Hinblick auf adaptive Berechnungen ist dieser Ansatz aufgrund der damit verbundenen Komplexität weniger praktikabel, weshalb dieser hier nicht weiter verfolgt wird. VIRTUALFLUIDS verfügt jedoch über einen solchen vektoroptimierten Kern (*LbVector*), in dem dieser Ansatz für a priori verfeinerte Gitter in 2-D umgesetzt wurde und mit dem diverse Benchmarkberechnungen durchgeführt wurden [47]. In dieser Arbeit wird der *LbVector*-Kern in Kap. 10 verwendet.

6.3 Gittergenerierung

Um die im Berechnungskern zu behandelnden Sonderfälle zu minimieren muss, die Gittertopologie bestimmte Kriterien erfüllen. Aus diesem Grund wurden im Vorfeld einige Bedingungen festgelegt, die ein gültiges Berechnungsgitter erfüllen muss. Um die Skalierung zu vereinfachen, dürfen sich zwei benachbarte Knoten um maximal einen Gitterlevel unterscheiden. Dies erreicht man indem man den Quad-/Octree nach dem Verfeinern (Abb. 6.7a) glättet (Abb. 6.7b). Um zusätzliche Raum- und Zeitinterpolationen während der Skalierung zu vermeiden, muss zudem gewährleistet sein, dass jeder für die kubische Rauminterpolation der hängenden Knoten benötigte Quellknoten entweder denselben Level aufweist oder maximal einen Level gröber ist. Diese Bedingung wird durch eine Levelweite von zwei oder höher erfüllt. Dadurch wird garantiert, dass jeder Knoten mindestens einen Nachbarn gleichen Levels in jede Richtung besitzt (Abb. 6.7c).



(a) *Basisoctree* ***(b)*** *Geglätteter Octree* ***(c)*** *Octree mit Levelweite 2*

Abbildung 6.7: *Schnitt durch Octree*

Um das Gitter zu generieren bzw. anzupassen, gibt es verschiedene Möglichkeiten. Der Anwender kann z. B. zu Beginn der Rechnung die Auflösung und Leveltiefe des uniformen Basisgitters definieren. Für weitere Änderungen stehen entsprechende Methoden der Gitterklasse zur Verfügung. Diese umfassen allerdings nur Basisfunktionalitäten wie die Verfeinerung bzw. Vergröberung eines Blockes. Bei allen weiterführenden Operationen wurde bewusst auf die Erweiterung der Klassendeklaration verzichtet, da eine solche dem objektorientierten Gedanken widersprechen würde. Stattdessen bietet das Gitter nach außen hin entsprechende Kontextschnittstellen an, die Objekte gemäß dem Besucher-Entwurfsmuster entgegen nehmen.

Das Besucher-Muster (*engl.: visitor pattern*) gehört zu den Verhaltensmustern. Es kapselt Operationen, die auf Elementen einer Objektstruktur ausgeführt werden. Neue Operationen können dadurch ohne Veränderung der betroffenen Elementklassen definiert werden.

Somit können z. B. topologische Änderungen mit Hilfe von sogenannten Besuchermodulen durchgeführt werden, die definierte Schnittstellen aufweisen. Diese werden dem Knotengitter übergeben, das dann die jeweiligen Algorithmen ausführt. Einer der wichtigsten Schnittstellen für derartige Module ist das INodeVisitor-Interface (Quellcode 6.1).

Quellcode 6.1: *Besucherinterfaceklasse* INodeVisitor

```
1  class INodeVisitor
2  {
3  public:
4     INodeVisitor(int startLevel, int stopLevel) { ... }
5     ~virtual INodeVisitor() { }
6
7     virtual bool isIterative() = 0;
8     virtual bool adaptNode( OctNodeGrid& grid, OctNode& node ) = 0;
9     /* ... */
10 };
```

Im Gegensatz zu anderen Programmiersprachen, wie Java oder C#, existiert in C++ kein eigenständiger Datentyp für Interfaceklassen. Stattdessen werden diese dort mit Hilfe von rein abstrakten Klassen definiert. Eine Besucherklasse, die das Interface INodeVisitor implementiert und an das Knotengitter übergeben wird, führt die in der Methode adaptNode definierten Operationen für jeden Knoten *node* der Level *startLevel* bis einschließlich *stopLevel* des Knotengitters *grid* durch. Zudem kann angegeben werden, ob es sich um einen iterativen Algorithmus handelt, der eventuell mehrfach durchgeführt werden muss. Ein einfaches Beispiel für eine derartige Besucherklasse ist das Verfeinerungsmodul RefineVisitor. Diese verfeinert beispielsweise alle Gitterknoten innerhalb einer vorgegebenen Geometrie. Vereinfacht sieht das Modul folgendermaßen aus:

Quellcode 6.2: *Besucherklasse* RefineVisitor

```
1  class RefineVisitor : public INodeVisitor
2  {
3  public:
4     RefineVisitor( GbObject geoObject, int startLevel, int stopLevel )
5       : INodeVisitor( startLevel,stopLevel)
6        , geoObject( geoObject )
7     {
8     }
9
10    virtual bool isIterative()
11    {
12       return true;
13    }
14
15    bool adaptNode( OctNodeGrid& grid, OctNode& node )
16    {
17       double x1w = grid.getCoordTrafo().getWorldX1( node );
18       double x2w = grid.getCoordTrafo().getWorldX2( node );
19       double x3w = grid.getCoordTrafo().getWorldX3( node );
20
21       if( geoObject.isPointInGbObject3D( x1w, x2w, x3w ) )
22       {
23          grid.expandNode( node );
24          return true;
25       }
26
27       return false;
28    }
29
30 private:
31    GbObject geoObject;
32 };
```

Die Verfeinerung des Knotengitters innerhalb einer Kugel kann beispielsweise wie folgt durchgeführt werden:

Quellcode 6.3: *Anwendungsbeispiel für den RefineVisitor*

```
1 int main()
2 {
3     OctNodeGrid grid;
4     /*...*/
5     GbSphere3D sphere( 10/*x1*/, 10/*x2*/, 10/*x3*/, 5/*rad*/ );
6     RefineVisitor refine( sphere, 0/*startLevel*/, 3/*stopLevel*/ );
7     grid.adaptByVisitor( refine );
8     /*...*/
9 }
```

Das Muster ist für beliebige Zwecke einsetzbar. Weitere Beispiele sind der InitializeDistributions- und NodeRatioVisitor, die zum Initialisieren der Knotenverteilungen bzw. zur Gitterglättung verwendet werden.

An dieser Stelle sei noch ein weiteres (auch für die Gittergenerierung) wichtiges Instrument erwähnt: der Interactor. Mit diesem umfangreichen Kopplungsmodul können beliebige Geometrieobjekte auf das Berechnungsgitter abgebildet werden. Bei Festkörpern werden dabei die inneren Knoten deaktiviert und unmittelbar an die Geometrie angrenzende Knoten mit Randbedingungen (Flags, **q**, etc.) versehen. Darüber hinaus finden Interaktoren auch bei der Gitterverfeinerung entlang von Körperoberflächen Anwendung. Auf die genaue Funktionsweise von Interaktoren wird bei der Besprechung der hybriden Blockgitter in Kap. 16 eingegangen.

6.4 Simulationsablauf

Bevor VIRTUALFLUIDS gestartet werden kann, muss der Anwender den jeweiligen Testfall generieren. Hierzu definiert er das Basisgitter des gewünschten Strömungsmodells, die zu verwendenden Geometrien sowie die zugehörigen Randbedingungen. Hierzu stehen ihm eine Vielzahl vorgefertigter Module zur Verfügung. Anschließend kann er die Verfeinerungszonen definieren und die notwendigen Gittermodifikationen durchführen. Nach dem Festlegen der Strömungsparameter kann die Simulation gestartet werden. Zum Abschluss des Präprozesses werden zu Beginn der Berechnungsmethode die Deskriptoren, die notwendigen Knotenflags und die für die Berechnung notwendige Datenstruktur initialisiert. Die Berechnungsschleife für ein a priori verfeinertes Gitter erfolgt gemäß dem in Abb. 6.8 für verschiedene Strömungsmodelle dargestellten Schema.

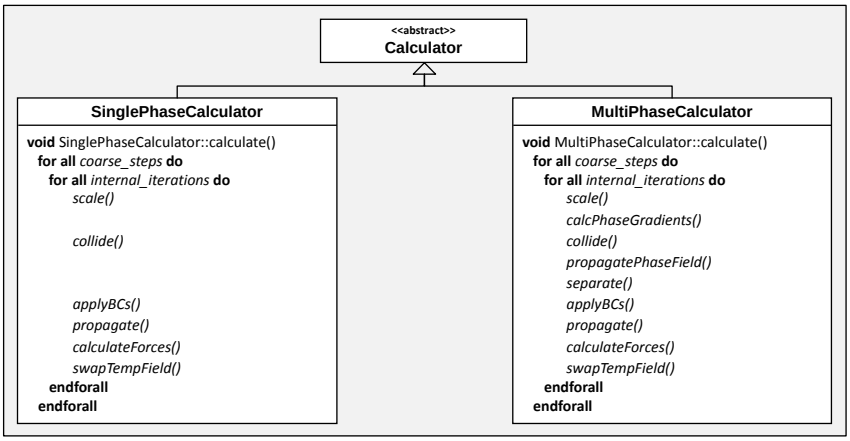


Abbildung 6.8: *Berechnungsroutine für Lattice-Boltzmann-Simulationen (Ein- und Mehrphase)*

7 Validierung I (Einphase, seriell)

Der 2-D-Knotengitterkern für Einphasenprobleme wurde umfangreich in [46] und [47] validiert. In diesem Kapitel werden verschiedene Ergebnisse für den 3-D-Code auf Basis des *d3q19*-Modells präsentiert.

Die in den folgenden Abschnitten zur Bestimmung der dimensionslosen Kennzahlen benötigte Kraft $\mathbf{F}_{Geo}$ aus dem Fluidgebiet erfolgt mit der Impulsaustauschmethode [110]:

$$\mathbf{F}_{Geo} = \sum_{Randknoten} \frac{\Delta x_l^{Dimension}}{\Delta t_l} \sum_{i=1}^{q} \mathbf{e}_i \left(f_i(t,\mathbf{x}) + f_{inv}(t+\Delta t_l,\mathbf{x}) \right) \tag{7.1}$$

Hierbei werden sämtliche Teilimpulse, die während der Propagation mit dem Festkörper ausgetauscht werden, aufsummiert (Abb. 7.1). Die Kraftberechnung erfolgt jeweils nach der Kollision und der Behandlung der Randbedingungen. Bei Verwendung der Simple-Bounce-Back-Randbedingung oder bei einem normierten Abstand von 0,5 sind $f_i(t,\mathbf{x})$ und $f_{inv}(t+\Delta t_l,\mathbf{x})$ identisch (vgl. Abschn. 3.2).

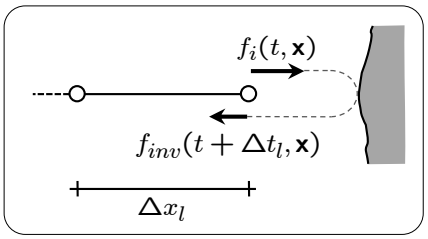


Abbildung 7.1: *Impulsaustausch*

7.1 Stationäre Spaltströmung (Hagen-Poiseuille)

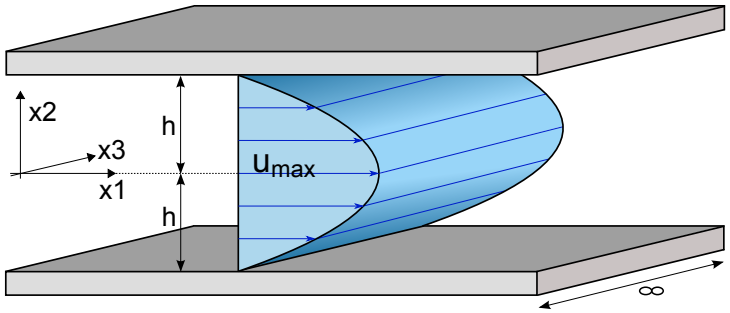


Abbildung 7.2: *Hagen-Poiseuille-Spaltströmung*

Die Hagen-Poiseuille-Strömung ist ein oft verwendeter Testfall, da sich diese durch ein relativ einfaches Setup auszeichnet (Abb. 7.2) und für diese eine analytische Lösung existiert. Für Strömungen

zwischen zwei ebenen, unendlich ausgedehnten Platten stellt sich im stationären Zustand eine laminare Schichtenströmung mit parabolischem Profil mit der Geschwindigkeit u_{max} im Scheitelpunkt ein. Folgende Parameter sind hier von Bedeutung [158]:

$$Re = \frac{u_m \, 2h}{\nu} \tag{7.2}$$

$$u_m = \frac{2}{3} u_{max} \tag{7.3}$$

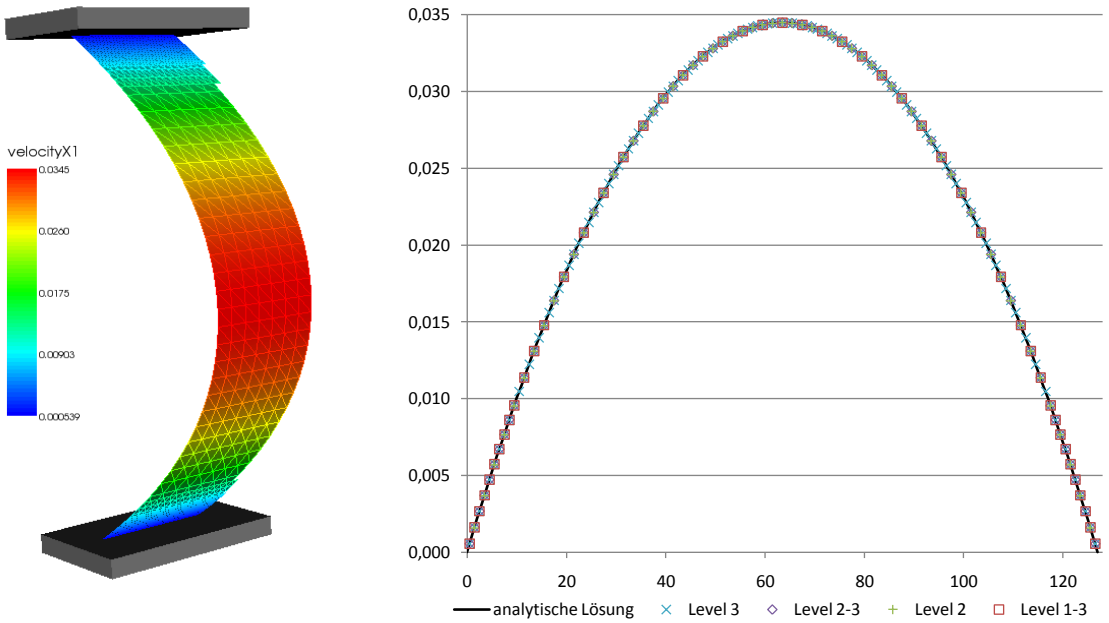
$$u_{max} = \frac{h^2}{2\,\nu} \frac{dp}{dx_1} \quad \text{mit} \quad \frac{dp}{dx_1} = F_{x_1} = \text{const} \tag{7.4}$$

$$u(x_2) = u_{x_1} \frac{(x_2 + h)\,(x_2 - h)}{h^2} \tag{7.5}$$

Der Druckgradient $\frac{dp}{dx_1}$ wird durch eine konstante Volumenkraft F_{x_1} (vgl. Abschn. 3.2) ersetzt. Das Gebiet ist in x_1- und x_3-Richtung periodisch. Simulationen mit einer Kanalhöhe von 127 Δx im feinsten Level (=Level 3) lieferten für unterschiedliche Gitterauflösungen, bei denen das Fluid in Gebietsmitte vergröbert wurde (Abb. 7.3a), Berechnungsergebnisse, die sehr gut mit der analytischen Lösung überstimmen (Tab. 7.1 und Abb. 7.3b).

h	Δx_{fein}	Level	ν	F_{x_1}	$u_{max,\,analyt}$	$u_{max,\,num}$	rel. Fehler [%]
63,5	2,0	2-2	0,05842	9,99685E-7	0,0345	0,0344815992	0,05334
63,5	1,0	1-3	0,05842	9,99685E-7	0,0345	0,0344977120	0,00631
63,5	1,0	2-3	0,05842	9,99685E-7	0,0345	0,0344977120	0,00631
63,5	1,0	3-3	0,05842	9,99685E-7	0,0345	0,0344977123	0,00630

Tabelle 7.1: *Ergebnisse für Hagen-Poiseuille-Strömung mit Re=50 und* $\Delta x_{Level\ 3} = 1$



(a) *Überhöhte Darstellung von* u_{x_1} *(Level 1-3)* ***(b)*** u_{x_1} *über Spalthöhe für verschiedene Auflösungen*

Abbildung 7.3: *Spaltströmung für Re=50*

7.2 Vollausgebildete laminare Rohrströmung

Bei laminaren Rohrströmungen von Newton'schen Fluiden lässt sich die Lösung ebenfalls analytisch ermitteln. Aufgrund der Haftrandbedingung verschwindet die Geschwindigkeit an der Rohrwand $\big(u(r=R)=0\big)$. Zudem ist bei einer vollausgebildeten Rohrströmung der Druckgradient $\frac{dp}{dx_1}$ konstant und entspricht der anzusetzenden Kraft F_{x_1}. Die Geschwindigkeiten für ein Rohr mit Radius R ergeben sich zu [158]:

$$u_{max} = \frac{R^2}{4\,\nu}\frac{dp}{dx_1} \quad \text{mit} \quad \frac{dp}{dx_1} = F_{x_1} = \text{const} \tag{7.6}$$

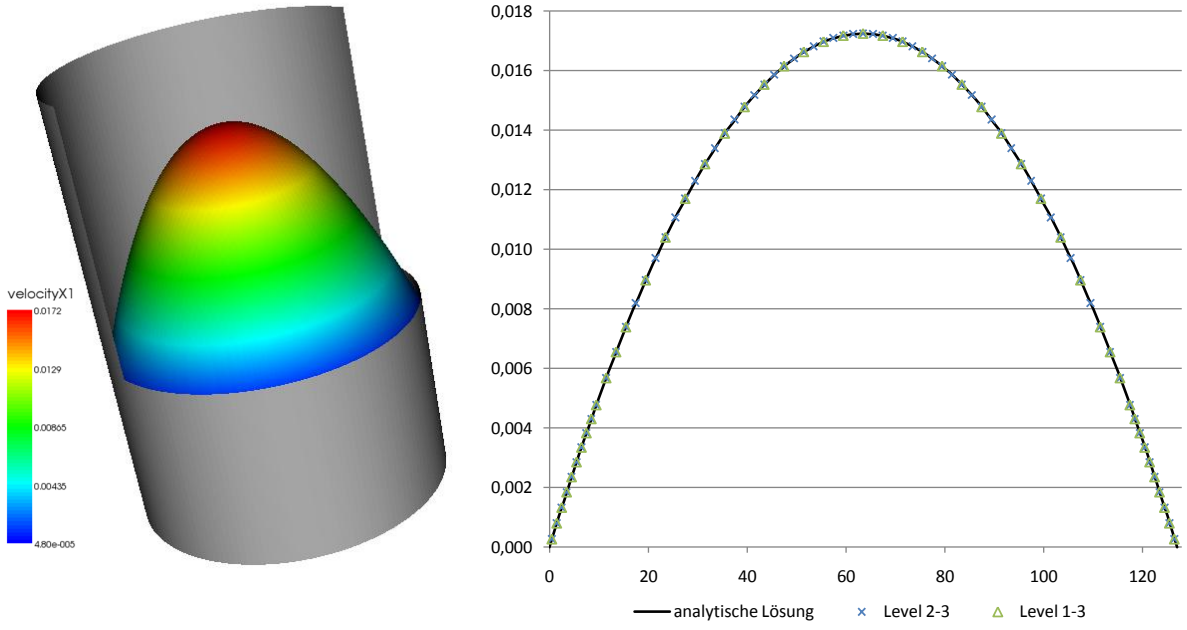
$$u_m = \frac{1}{2}u_{max} \tag{7.7}$$

$$u(r) = \frac{u_{max}}{4\,\nu}\left(R^2 - r^2\right) \tag{7.8}$$

Das Geschwindigkeitsprofil ist somit ein Rotationsparaboloid mit Scheitelpunkt in der Rohrachse $\big(u(r=0)=u_{max}\big)$ (Abb. 7.4a). Die Reynoldszahl wird mit $Re = \frac{u_m\,2\,R}{\nu}$ bestimmt. Das Gebiet ist in x_1-Richtung periodisch und an der Zylinderwand kommt die Boundary-Fitting-Randbedingung zur Anwendung. Die numerische Lösung stimmt hier gut mit der analytischen überein (Tab. 7.2 und Abb. 7.4b).

R	Δx_{fein}	Level	ν	F_{x_1}	$u_{max,\,analyt}$	$u_{max,\,num}$	rel. Fehler [%]
63,5	1,0	3-3	0,05842	9,99685E-7	0,017250	0,01724798	0,01169
63,5	1,0	2-3	0,05842	9,99685E-7	0,017250	0,01724798	0,01170
63,5	1,0	1-3	0,05842	9,99685E-7	0,017250	0,01724795	0,01188

Tabelle 7.2: *Rohrströmung mit Re=18,75*



(a) *Überhöhte Darstellung von u_{x_1} (Level 1-3)* ***(b)*** *Eu_{x_1} über Spalthöhe für verschiedene Auflösungen*

Abbildung 7.4: *Rohrströmung für Re=18,75*

Die nicht-uniformen Knotengitter wurden sowohl am Zylinderrand als auch in Gebietsmitte verfeinert, um den Scheitelpunkt entsprechend abbilden zu können. Die Unterschiede zwischen den Ergebnissen für verschiedene Gitterauflösungen sind vernachlässigbar. Man erhält hier somit bei partiell gröberer Gitterauflösung adäquate Lösungen zu den rein uniformen Gebieten und benötigt im Vergleich weniger Rechenressourcen.

7.3 Laminare Strömung um einen Zylinder

1996 veröffentlichten Schäfer und Turek in [159] diverse 2-D- und 3-D-Benchmarkkonfigurationen mit Ergebnissen von unterschiedlichen Navier-Stokes-Strömungslösern. Einer dieser Testfälle ist die Strömung um einen Zylinder, der sich waagerecht in einem Rechteckkanal befindet (Abb. 7.5).

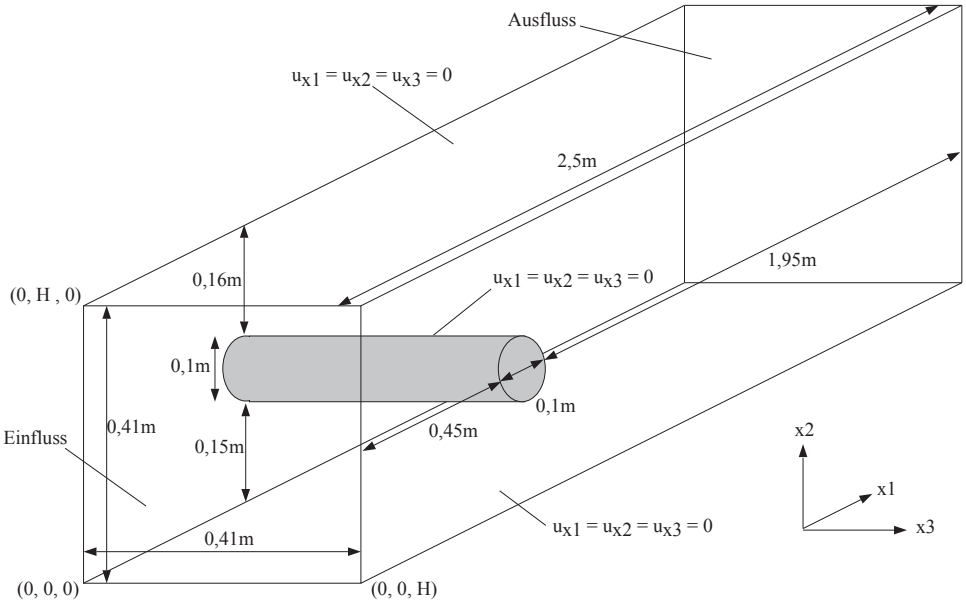


Abbildung 7.5: *Setup*

Für eine Reynoldszahl von 20 sind verschiedene Vergleichswerte, wie z. B. die dimensionslosen Kraftbeiwerte c_D (*engl.: drag*) und c_L (*engl.: lift*) des Zylinders mit Durchmesser D, angegeben. Die Berechnung dieser Kennzahlen ist für einen Kanal mit Höhe H definiert zu:

$$Re = \frac{u_{x_1,m}\, D}{\nu} \quad \text{mit} \quad u_{x_1,m} = \frac{4}{9}\, u_{x_1,max} \tag{7.9}$$

$$c_D = \frac{2\, F_{Zyl,x_1}}{\rho\, u_{x_1,m}^2\, D\, H} \tag{7.10}$$

$$c_L = \frac{2\, F_{Zyl,x_2}}{\rho\, u_{x_1,m}^2\, D\, H} \tag{7.11}$$

Das Geschwindigkeitseinstromprofil ist vorgegeben:

$$u_{x_1}(0, x_2, x_3) = 16\, u_{x_1,m}\, x_2\, x_3 \frac{(H - x_2)(H - x_3)}{H^4} \quad \text{und} \quad u_{x_2} = u_{x_3} = 0 \tag{7.12}$$

Am Ausfluss kommt eine Druckrandbedingung zum Einsatz und sowohl die Wände als auch der Zylinder werden mit einer Haftrandbedingung diskretisiert (*Boundary-Fitting*). Die Simulation wurde unter Verwendung des Momentenmodells durchgeführt. Die Verfeinerung für die Konfigurationen Level 0-1/2/3 erfolgte ausschließlich im Bereich des Zylinders. Bei der Simulation für Level 1-3 wurde zudem der Einfluss verfeinert (Abb. 7.6).

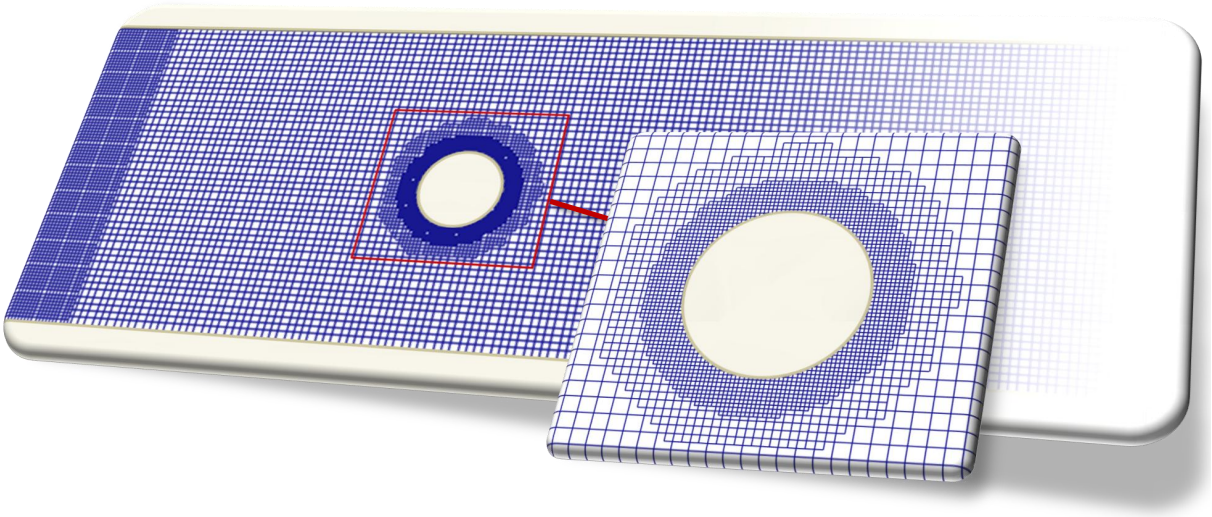


Abbildung 7.6: *Gebietsverfeinerung (Schnitt durch Gebietsmitte)*

Level	Knoten über Zylinderhöhe	c_D	c_L
0-0	5,55	7,02	0,2845
0-1	11,11	6,32	0,1361
0-2	22,22	6,24	0,0304
0-3	44,44	6,09	0,0099
1-1	11,11	6,32	0,0077
1-3	44,44	6,08	0,0082
Referenz-spektrum		6,05 6,10	0,008 0,100

Tabelle 7.3: *Ergebnisse für Zylinderumströmung mit Re=20*

Die Ergebnisse für $Re = 20$ und $u_{x_1,max} = 0,045$ sind in Tab. 7.3 dargestellt. Sie liegen im Referenzspektrum der Resultate der anderen Gruppen. Wie erwartet, sinkt mit steigender Grenzschichtauflösung der relative Fehler.

7.4 Schleichende Strömung normalviskoser Fluide

Bei dieser Art von Strömungen werden die Trägheitskräfte gegenüber den Zähigkeitskräften vernachlässigt. Voraussetzung hierfür sind kleine Reynoldszahlen (im Allgemeinen $Re \leq 1$). Ein bekanntes Beispiel hierfür ist die Stokesströmung um eine Kugel. Um Einflüsse durch die Randbedingungen zu vermeiden, sollte das Verhältnis der Gebietslänge H zum Kugeldurchmesser D größer 320 sein [22].

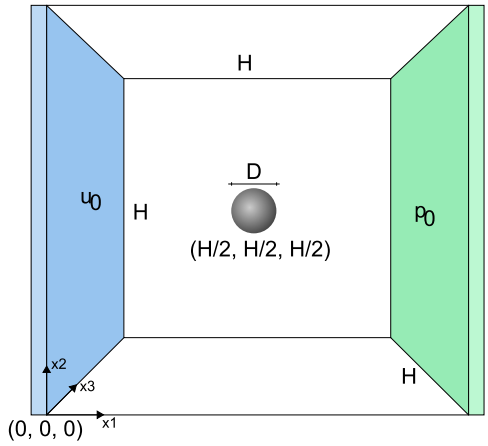


Abbildung 7.7: *Setup für Stokesströmung um eine Kugel*

Folgende Kennzahlen sind nach [158] für dieses Problem maßgeblich :

$$Re = \frac{u_{x_1} D}{\nu} \tag{7.13}$$

$$c_D = \frac{8 F_{Kugel}}{\rho u_{x_1}^2 \pi D^2} \tag{7.14}$$

$$c_{D,Stokes} = \frac{24}{Re} \tag{7.15}$$

$$c_{D,Oseen} = \frac{24}{Re}\left(1 + \frac{3}{16} Re\right) \tag{7.16}$$

Die Kraftbeiwerte nach Stokes und Oseen gelten ausschließlich für viskose Öle oder Fluide mit geringer Viskosität und für sehr kleine Kugeldurchmesser D (z. B. winzige Nebeltröpfchen in der Atmosphäre). Die Korrektur von Oseen (Gl. 7.16) erlaubt eine maximale Renyoldszahl von fünf [158].

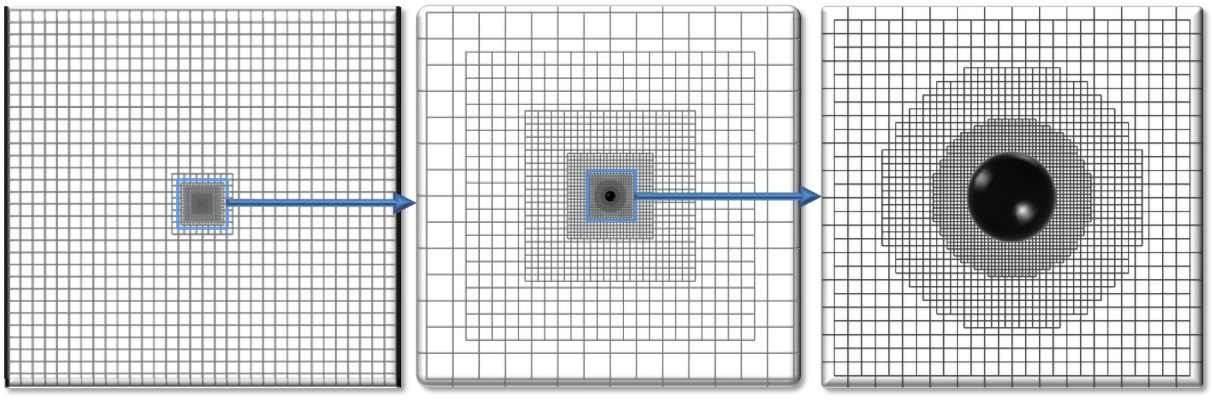


Abbildung 7.8: *Simulationssetup Level 0-8 (Schnitt durch Gebietsmitte)*

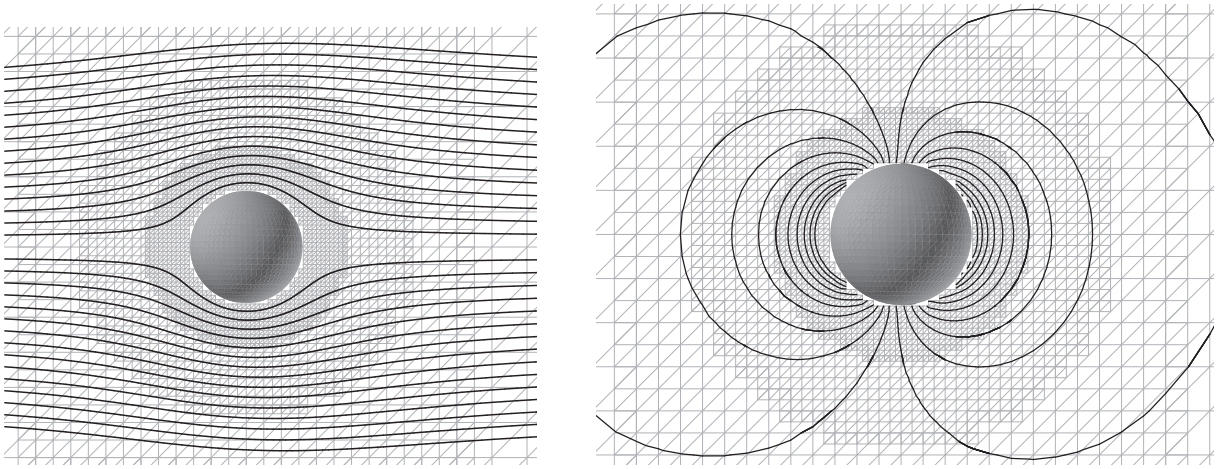
Die Simulation mit VIRTUALFLUIDS erfolgte auf einem nicht-uniformen Gitter, das im Bereich der Kugel verfeinert wurde (Abb. 7.8). Am Einfluss wurde ein konstantes Geschwindigkeitsprofil mit $u_0 = 3{,}26 \cdot 10^{-3}$ vorgegeben. Am Ausfluss wurde ein konstanter Druck p_0 angesetzt. In die anderen Richtungen war das Gebiet periodisch. Für eine Simulation mit einer Kugelauflösung von 25 (50)

Knoten über die Kugelhöhe benötigt ein gleichwertiges, uniformes Gitter $5,4 \cdot 10^{11}$ ($4,5 \cdot 10^{12}$) Berechnungsknoten und kann somit nicht mehr auf herkömmlichen Systemen gerechnet werden. Erst die Verwendung lokaler Gitterverfeinerung macht eine effiziente Berechnung dieses Problems möglich. Die hier verwendeten Gitter erforderten für den Testfall *Level 0-8* $1,5 \cdot 10^5$ und für *Level 0-9* $3,5 \cdot 10^5$ Gitterknoten und konnten auf einem handelsüblichen PC berechnet werden.

Level	D (Knoten)	$c_{D,Stokes}$	$c_{D,Oseen}$	$c_{D,num}$	rel. Fehler [%]
0-8	25	24,00	25,13	28,84	14,77
0-9	50	24,00	25,13	24,51	3,30

Tabelle 7.4: *Kennzahlen für Kriechströmung bei Re=1*

Die Lösung stimmt gut mit den Abschätzungen von Oseen und Stokes überein (Tab. 7.4). Die Verwendung einer adaptiven Verfeinerung könnte hier weitere Erkenntnisse liefern. In zwei Raumdimensionen erzielte die LB-Simulation einer Stokesströmung um eine Kugel von Bernd Crouse auf Basis einer mit Divergenzsensor gesteuerten Gitterverfeinerung qualitativ genauere Ergebnisse gegenüber Berechnungen mit a priori verfeinerten Gittern [22].



(a) *Stromlinien* ***(b)*** *Isolinien des Drucks*

Abbildung 7.9: *Schnitt durch Gebietsmitte (Level 0-8)*

8 Erweiterung I - dynamisch adaptive Knotengitter

Bei komplexen Strömungsproblemen oder Simulationen mit kontinuierlich im Raum veränderlichen Strömungseigenschaften (Wirbelstraßen, aufsteigende Blasen, etc.) ist die optimale Gitterstruktur a priori nicht abzuschätzen. Um für derartige Probleme die bestmögliche räumliche und zeitliche Gitterauflösung des Eulergitters zu gewährleisten, stellt die Verwendung einer dynamisch adaptiven, also einer sich zur Laufzeit ändernden Gitterstruktur eine gute Lösung dar. Bei dieser als h-Adaptivität bezeichneten Vorgehensweise wird die Ergebnisqualität durch eine Anpassung des Netzes erreicht und nicht durch eine höhere Ordnung der Ansatzfunktion (p-Adaptivität).

Die Implementierung eines solchen Gitters ist relativ aufwändig, weshalb auch hier auf eine allgemeingültige Implementierung geachtet wurde, die somit wiederum allen Strömungslösern in VIRTUALFLUIDS zur Verfügung steht. Mittels (benutzer)definierter Adaptionskriterien wird zum einen eine optimale Gitterstruktur und zum anderen die Minimierung des Speicherbedarfs und der Rechenzeit gewährleistet. Besondere Beachtung wurde auch der Speicherverwaltung geschenkt, denn wie bei C ist auch im C++-Standard keine automatische Speicherverwaltung (*engl.: garbage collection*) enthalten. Somit muss sichergestellt werden, dass der Speicher beim Löschen von dynamisch allokierten Objekten auch wieder freigegeben wird.

8.1 Allgemeines

Im Vorfeld wurden verschiedene Bedingungen festgelegt, die bei der Adaptivitätssteuerung zu beachten sind. Um zusätzliche Zeit- und Rauminterpolationen zu vermeiden, soll die Gitteranpassung ausschließlich im gröbsten Zeitschritt und nicht innerhalb der gestaffelten Zeitschleife erfolgen (Abb. 8.1).

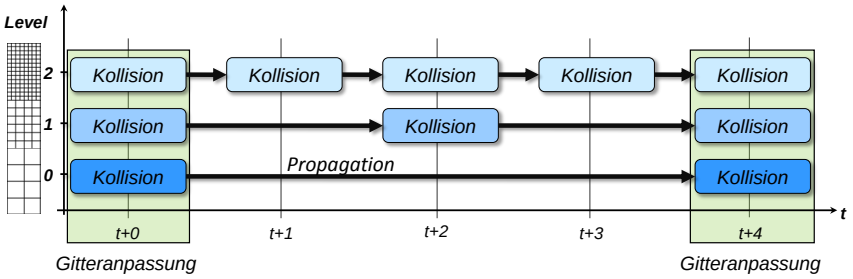


Abbildung 8.1: *Zeitpunkt der adaptiven Gitteranpassung*

Zudem muss gewährleistet sein, dass nach jedem Adaptionschritt die für eine LB-Berechnung gültige Gittertopologie vorliegt und die Datenstruktur der Berechnungsklasse aktualisiert wurde. Hierfür muss das Gitter geglättet und sämtliche Knotenflags sowie die normierten Abstände **q** und die Nachbarschaftsverhältnisse aktualisiert werden (vgl. Abschn. 6.3). Während der Umsetzung hat sich

herausgestellt, dass es u. a. zur Bewahrung der Symmetrie von Vorteil ist, die Gitterglättung bei jeder einzelnen Knotenverfeinerung/-vergröberung beizubehalten.

Dem Konzept entsprechend wird die physikalische Wertzuweisung in den speziellen Modulen der Physikschicht durchgeführt und die topologische Anpassung in der Topologieschicht. Für die bestehenden Löser sind somit außer der Definition der physikalischen Wertzuweisung und dem Aufruf der entsprechenden Methoden in der Zeitschleife keine weiteren Änderungen gegenüber Berechnungen auf a priori verfeinerten Gittern notwendig.

8.2 Änderung der Topologie

Bei der Verfeinerung/-gröberung eines Quad-/Octreeknotens werden bis zu 9/27 neue feine Knoten erzeugt bzw. gelöscht (Abb. 8.2).

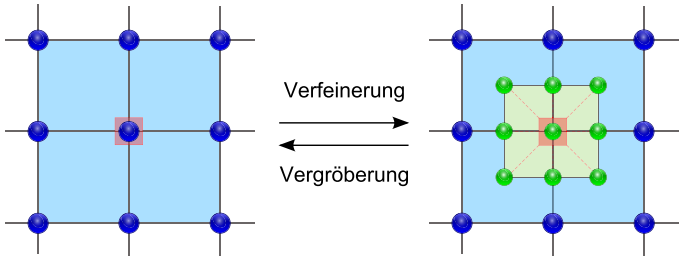


Abbildung 8.2: *Topologieänderung des Knotengitters*

Um die Gültigkeit der Quad-/Octreestruktur zu gewährleisten, gilt folgende Bedingung:

> Alle durch *eine* Verfeinerung generierten Knoten werden durch *eine* Vergröberung wieder entfernt

Aus Übersichtsgründen wird im Folgenden die Vorgehensweise der Netzanpassung nur für den Quadtreecode beschrieben. Beim Octreelöser finden vergleichbare Algorithmen unter Berücksichtigung der hinzukommenden Sonderfälle Anwendung.

8.2.1 Verfeinerung

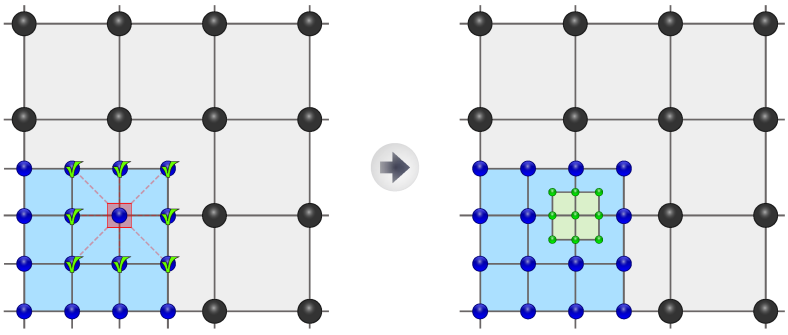


Abbildung 8.3: *Verfeinerung - Knotentyp I*

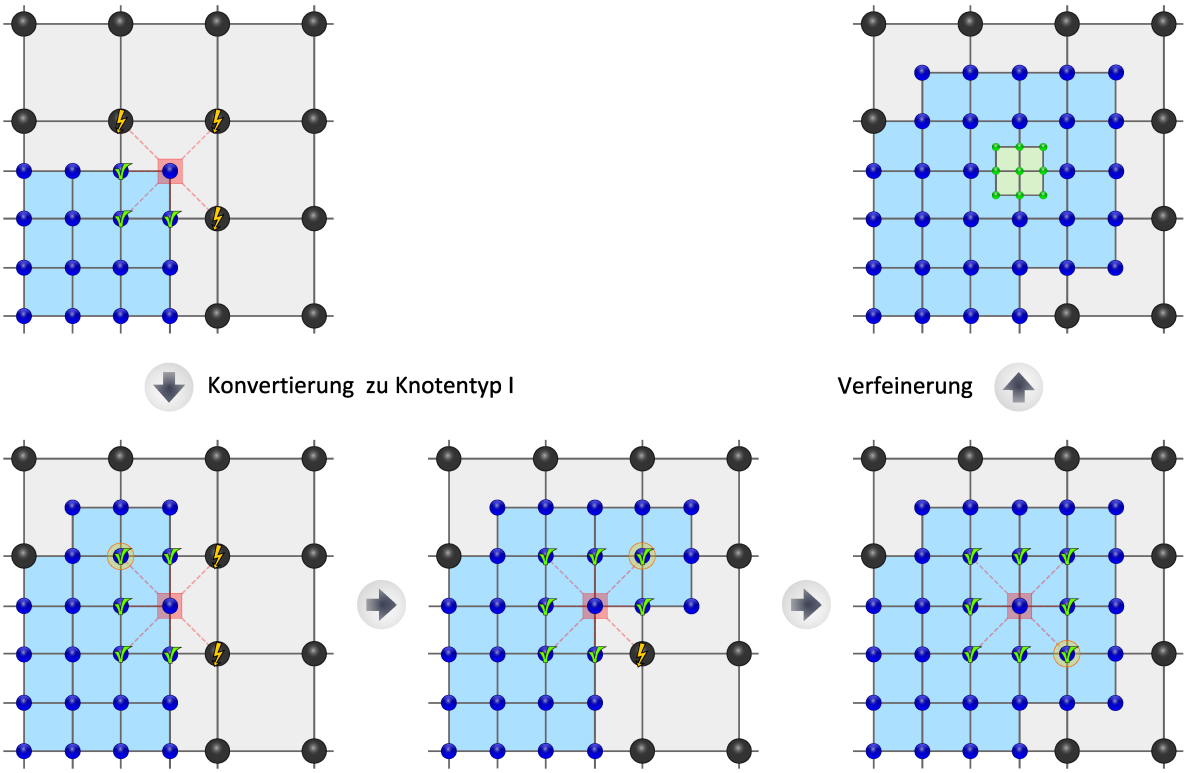


Abbildung 8.4: *Verfeinerung - Knotentyp II*

Algorithmus 2 Verfeinerungsroutine des Knotengitters (2-D)

```
function EXPANDNODE( node )
    if node.level() ≥ grid.finestLevel() then
        grid.initLevel( node.level()+1 )
    end if
    for all dirs in r_i mit i = 0..7 (3-D: i = 0,...,25) do
        neighNode = grid.getNode( node.indices().toNeigh( dir ) )
        if neighNode.exists() and neighNode.level() < node.level() then
            grid.expandNode( neighNode )
        end if
    end for
    grid.createNode( node.indizes().toFineLevel() )
    for all dirs in r_i mit i = 0..7 (3-D: i = 0,...,25) do
        neighNode = grid.getNode( node.indizes().toFineLevel().toNeigh( dir ) )
        if not neighNode.exists() then
            grid.createNode( node.indizes().toFineLevel().toNeigh( dir ) )
        end if
    end for
    if node.hasDescriptor() then
        node.descriptor().assignValuesToExpandedNodes( newNodes );
    end if
    grid.delete( node )
end function
```

Hierbei wird zwischen zwei Fällen unterschieden. Bei Knotentyp I befinden sich alle Nachbarknoten des zu verfeinernden Knotens mit Level L im Abstand $r_i \cdot \Delta x_L$ (vgl. Abschn. A3) im selben oder bereits im feineren Level (Abb. 8.3). Dadurch können die neuen Knoten dem Gitter sofort ohne weitere Änderungen hinzugefügt werden, sofern diese noch nicht existieren.

Weist einer der Nachbarknoten einen gröberen Level auf, so handelt es sich bei dem zu verfeinernden Knotens mit Level L um den Knotentyp II. In diesem Fall werden zuerst die groben Nachbarknoten rekursiv verfeinert, um das gewünschte Levelverhältnis zu bewahren. Dies entspricht einer Transformation zu Knotentyp I, der anschließend wie zuvor beschrieben behandelt werden kann. Der Pseudocode der implementierten Verfeinerungsroutine ist in Alg. 2 gegeben.

8.2.2 Vergröberung

Eine Vergröberung kann nur für Knoten mit Level > 0 durchgeführt werden, an deren Position auch ein grober Knoten initialisiert werden kann. Diese Voraussetzung ist für alle Knoten erfüllt, die ausschließlich gerade Knotenindizes besitzen. Andernfalls handelt es sich um einen hängenden Knoten, an dessen Position sich zur Bewahrung eines gültigen Netzes kein grober Knoten befinden darf (vgl. Abschn. 6.1). Um die Baumstruktur zu erhalten, werden die Nachbarknoten ermittelt, die zusätzlich gelöscht werden müssen. Hierzu wird, ausgehend von einem geglätteten Gitter, bei der Vergröberung eines Knotens mit Level L die vorhandene Leveldifferenz zu den Nachbarknoten im Abstand Δx_{L-1} kontrolliert. Hierbei sind bezüglich des Nachbarlevels zwei Fälle zu berücksichtigen:

1. *Nachbarlevel* = L:
   Der Nachbar befindet sich im selben Level. Somit darf keiner der Nachbarknoten im Abstand Δx_L gelöscht werden (vgl. Abb. 8.6b und Abb. 8.6c).

2. *Nachbarlevel* > L:
   Der Nachbar befindet sich in einem feineren Gitterlevel und muss zuvor analog zu Knotentyp II bei der Verfeinerung rekursiv vergröbert werden (Abb. 8.5).

Da sich an einer Position im Raum stets nur ein Knoten befinden darf, wird auch der Ausgangsknoten nach seiner Vergröberung gelöscht. Der Vergröberungsalgorithmus wurde durch Austausch der entsprechenden Methoden von der Verfeinerung übernommen.

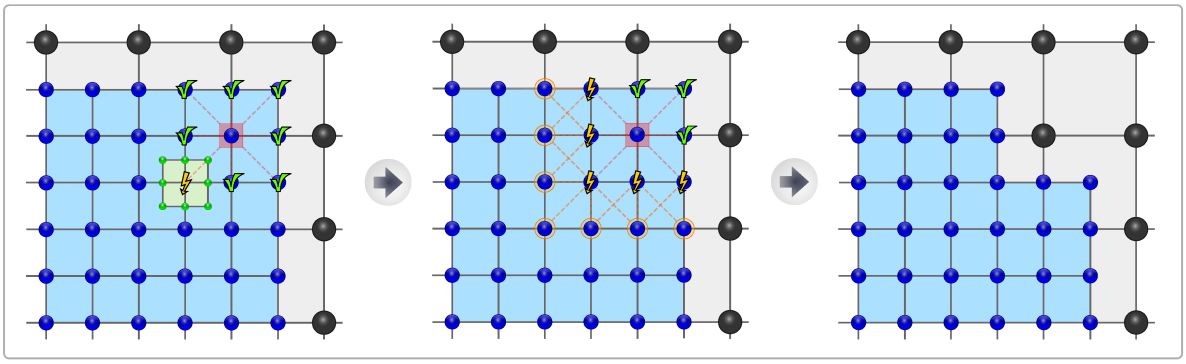
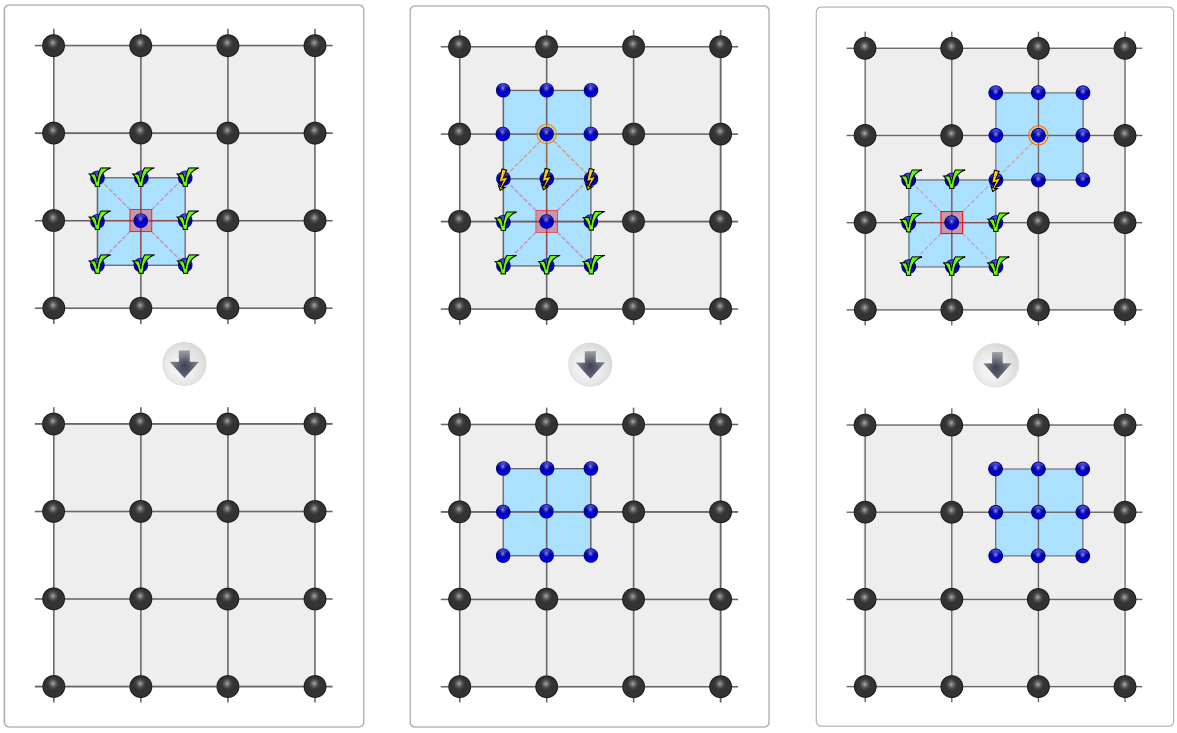


Abbildung 8.5: *Vergröberung - Knotentyp II*



(a) *(b)* *(c)*

Abbildung 8.6: *Vergröberung - Knotentyp I*

8.3 Werteinterpolation der Knotendeskriptoren

Für die Zuweisung der physikalischen Knotenattribute müssen im speziellen Knotendeskriptor die abstrakten Methoden assignValuesToExpandedNodes() und assignValuesToCollapsedNodes() des NodeDescriptors implementiert werden. Zu deren Argumente gehören u. a. die Referenz auf das Gitter und die neu erzeugten Knoten.

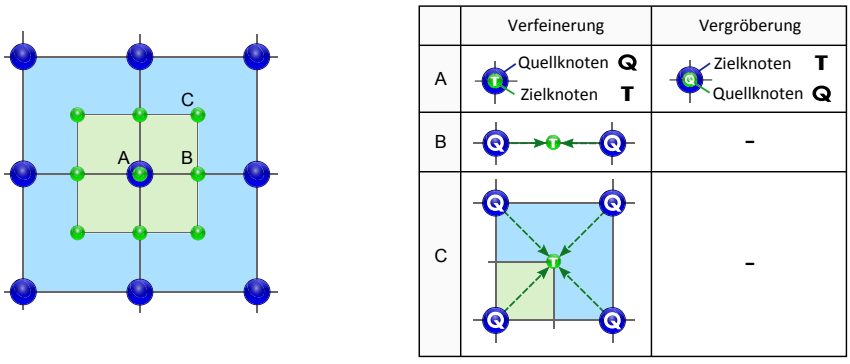


Abbildung 8.7: *Knoteninterpolationen bei Wertzuweisung*

In den Methoden der LB-Deskriptoren werden zuerst die zum Zielknoten T zugehörigen Quellknoten Q ermittelt (Abb. 8.7) und anschließend die Partikelverteilungen f_i optional wie folgt zugewiesen:

1. Die Verteilungen der Quellknoten werden wenn nötig in den Ziellevel skaliert (vgl. Abschn. 3.3). Je nach Knotentyp folgt dann ggf. eine räumliche (bi)lineare Interpolation dieser Knotenwerte, die in diesem Fall einer Mittelwertbildung entspricht:

$$f_{i,Level\,T}(t,\mathbf{x}_T) = \frac{1}{n}\sum_{i}^{n} f_{i,Level\,T}\left(t,\mathbf{x}_Q\right) \tag{8.1}$$

mit

$$f_{i,Level\,T}\left(t,\mathbf{x}_Q\right) = f_i^{eq}\left(t,\mathbf{x}_Q\right) + s_{Level\,Q\to T}\; f_{i,Level\,Q}^{neq}\left(t,\mathbf{x}_Q\right)$$

2. Die Bestimmung der Gleichgewichtsverteilung des Zielknotens erfolgt nicht durch Interpolation der Quellgleichgewichtsverteilungen, sondern wird mit Hilfe der rauminterpolierten makroskopischen Größen am Zielknoten berechnet. Die Nichtgleichgewichtsverteilungen werden wie in Gl. 8.1 berechnet und anteilig addiert:

$$f_{i,Level\,T}(t,\mathbf{x}_T) = f_i^{eq}\left(\,\overline{\rho}\,(t,\mathbf{x}_T)\,,\,\overline{\mathbf{u}}(t,\mathbf{x}_T)\,\right) + \frac{1}{n}\sum_{i}^{n} s_{Level\,Q\to T}\; f_{i,Level\,Q}^{neq}\left(t,\mathbf{x}_Q\right) \tag{8.2}$$

mit

$$\begin{aligned}\overline{\rho}(t,\mathbf{x}_T) &= \frac{1}{n}\sum_{i}^{n}\rho\left(t,\mathbf{x}_Q\right)\\ \overline{\mathbf{u}}(t,\mathbf{x}_T) &= \frac{1}{n}\sum_{i}^{n}\mathbf{u}\left(t,\mathbf{x}_Q\right)\end{aligned}$$

Ungültige Nachbarn, wie z. B. fehlende oder nicht-aktive Knoten, gehen nicht in die Interpolation mit ein. Beim Momentenmodell können anstelle der Verteilungen die entsprechenden Momente skaliert und interpoliert werden.

Die zusätzlichen Eigenschaften der Mehrphasendeskriptoren sind levelunabhängig (ϕ, ψ, T). Deshalb werden sie wenn nötig interpoliert und anschließend ohne Skalierung zugewiesen.

8.4 Vorausschauende Gitteranpassung

Die Gitteradaption erfolgt zu definierten Zeitpunkten im Anschluss an die gestaffelte Zeitschleife. In Abb. 8.8 ist die zugehörige Erweiterung der Berechnungsschleife dargestellt. Anhand einer aufsteigenden Blase (Abb. 8.10a) wird im Anschluss schematisch die prinzipielle Vorgehensweise eines Adaptionsschrittes erläutert.

Ziel bei der Mehrphasenberechnung soll es sein, das Phaseninterface fortlaufend mit dem höchsten Gitterlevel zu diskretisieren. Die Simulation des Phasenfeldes außerhalb dieses Übergangsbereiches kann in verschiedenen Gitterleveln erfolgen. Um dies zu gewährleisten, wird in jedem Adaptionsschritt zunächst die voraussichtliche Bewegung des Interfaces bis zur nächsten Anpassung abgeschätzt (Abb. 8.9a) und der ermittelte Bereich im Anschluss verfeinert (Abb. 8.9b).

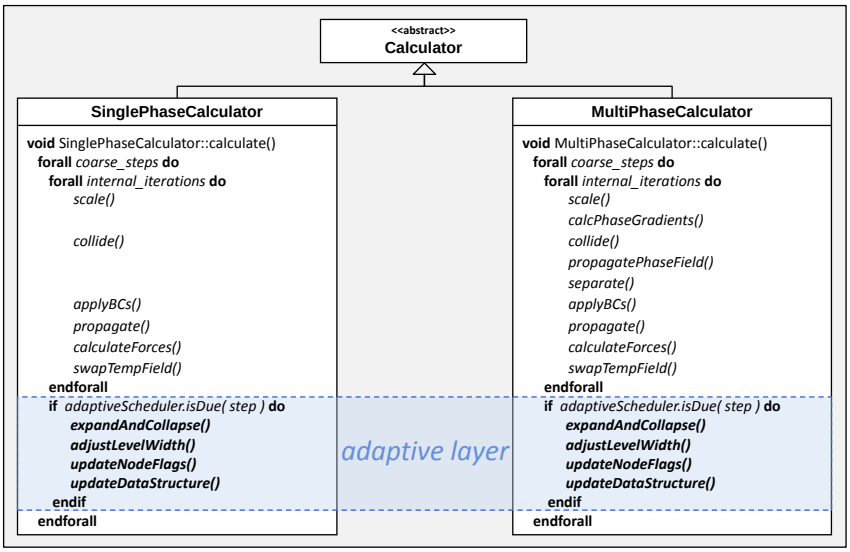
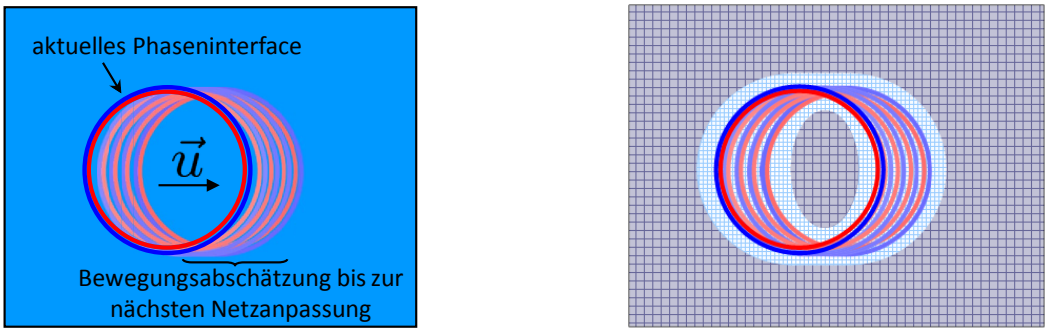


Abbildung 8.8: *Erweiterte Berechnungsroutine für adaptive Simulationen (Ein- und Mehrphase)*



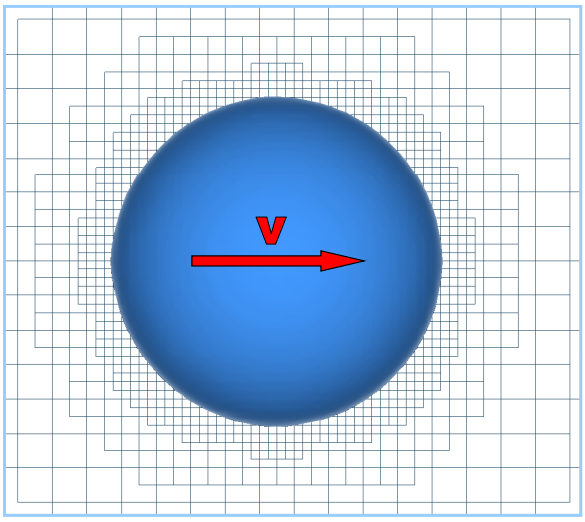
(a) *Abschätzung der Phasenfeldbewegung* ***(b)*** *Minimal benötigtes Verfeinerungsgebiet*

Abbildung 8.9: *Bestimmung des nötigen Verfeinerungsbereichs*

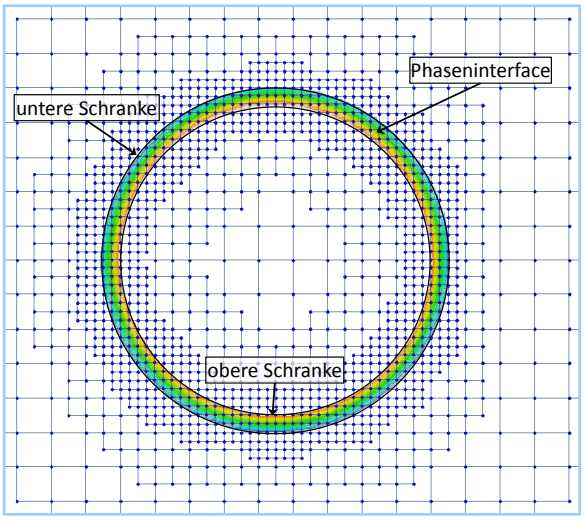
Für die Gitteranpassung wird die in Abschn. 6.3 beschriebene Knotenbesucherschnittstelle des Knotengitters verwendet. Die MPExpandVisitor-Klasse dient z. B. zur Sicherstellung des Gittervorlaufes. Sie überprüft, ob sich der Ordnungsparameter ϕ eines Knotens innerhalb des Phasenübergangs befindet (Abb. 8.10b). Liegt der Wert innerhalb eines vorgegebenen Bereiches, so wird er markiert, um später bei der Vergröberung ignoriert zu werden. Anschließend wird sichergestellt, dass die Entfernung des markierten Knotens zum Gitterinterface den geforderten Mindestabstand aufweist. Ist dies nicht der Fall, so wird entsprechend verfeinert (Abb. 8.10c). In diesem Schritt werden alle Knoten, die sich im feinsten Level befinden müssen, entsprechend markiert. Für eine effiziente Implementierung werden ausschließlich Positionen überprüft, an denen eine Vergröberung möglich ist.

Nachdem das MPExpandVisitor-Modul im Anschluss die nicht markierten Knoten vergröbert hat (Abb. 8.10d), wird in der Methode adjustLevelWitdh() der Berechnungsklasse (Abb. 8.8) die benötigte Levelweite (≥ 2, vgl. Abschn. 6.3) hergestellt (Abb. 8.10e). Bereits während der Vergröberung wird darauf geachtet, dass keine Knoten gelöscht werden, die hier wieder erzeugt werden müssten. Dadurch wird u. a. verhindert, dass wichtige Strömungsinformationen verloren gehen.

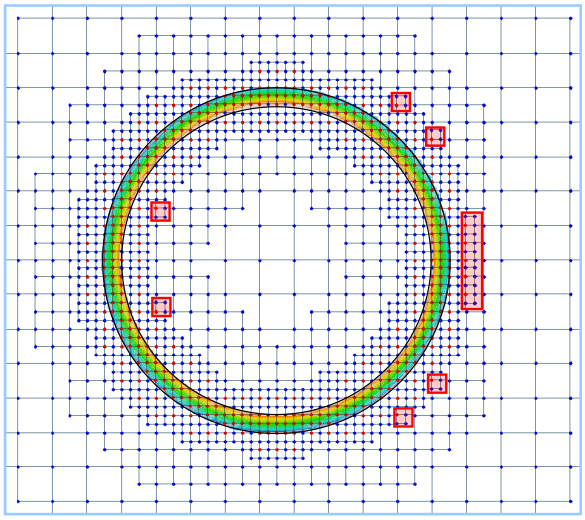
Die Methoden updateNodeFlags() und updateDataStructure() der Berechnungsklasse dienen zur Aktualisierung der Datenstruktur und somit zur Gewährleistung eines gültigen LB-Rechengitters. Hier



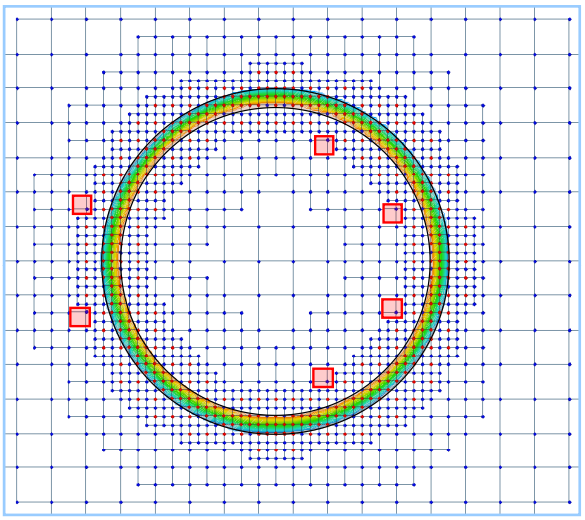
(a) *Ausgangssituation - Blase mit Geschwindigkeit* u_{x_3}



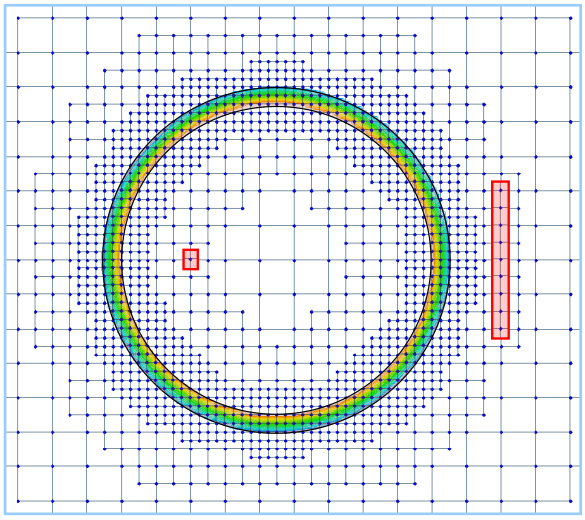
(b) *Phaseninterfacebereich (untere/obere Schranke)*



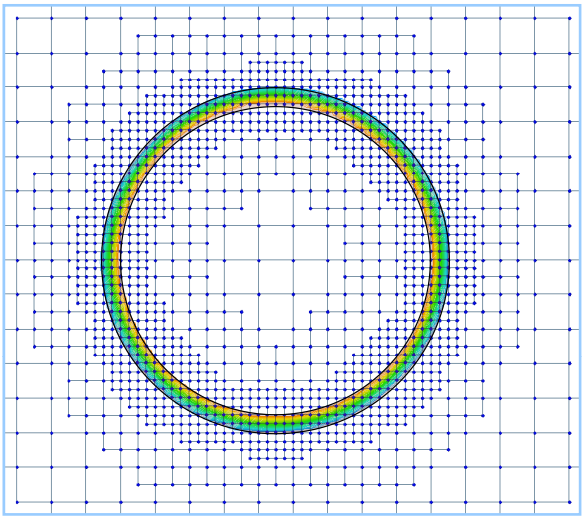
(c) *Knotenverfeinerung*



(d) *Knotenvergröberung*



(e) *Korrektur des Knotenlevelverhältnisses*



(f) *Gitter nach adaptiver Anpassung*

Abbildung 8.10: *Adaptive Gitterverfeinerung am Beispiel einer Blase in 3-D (Schnitt)*

werden die notwendigen Knotenflags sowie die Randbedingungsobjekte der Deskriptoren gesetzt und die Daten der Berechnungsklasse aufbereitet.

Damit ist die Netzanpassung abgeschlossen und die eigentliche Berechnung kann fortgesetzt werden.

Anmerkungen

Es ist zu beachten, dass der notwendige feine Gitterbereich zum einen von der Aufstiegsgeschwindigkeit der Blase als auch von der Anzahl der vorhandenen Gitterlevel abhängt. Mit steigender Levelzahl erhöht sich die Anzahl der internen Iterationschritte. Da eine Gitteränderung nur nach dem gröbsten Zeitschritt erfolgt, ist die Advektion des Phasenfeldes weiter fortgeschritten als bei geringerer Levelanzahl.

Die Netzanpassung erfolgt über allgemeingültige Instanzen der Knotenbesucherklassen Deshalb können diese jederzeit ausgetauscht oder miteinander kombiniert verwendet werden. Beim Einphasenkern können hierfür beispielsweise unterschiedliche Fehlerschätzer/-indikatoren implementiert werden. Die Verfeinerungstiefe kann abhängig von verschiedenen Schwellwerten gemacht werden. Beispiele für solche Indikatoren sind in [22] zu finden.

8.5 Teilfixierte Knotengitter

Um zu vermeiden, dass bestimme Regionen vergröbert werden, kann der Benutzer Regionen definieren, in denen eine gewisse Leveltiefe nicht unterschritten werden darf. Dies ist zum Beispiel bei der Abbildung von geometrischen Objekten von Bedeutung. Durch die sogenannte Levelfixierung wird verhindert, dass diese Objekte nach der automatischen Vergröberung nur noch unzureichend diskretisiert sind. In Abb. 8.11 ist als Beispiel ein an der Wand heruntergleitender Tropfen dargestellt, bei dessen Berechnung das in Abschn. 4.3 vorgestellte erweiterte Rothmann-Keller-Modell zur Anwendung kam. Im Bereich der Auswölbung soll die Leveltiefe von mindestens zwei gewährleistet sein. Der Tropfen selbst weist eine weitere Verfeinerungsstufe auf, sodass sich beim Überschreiten der Wölbung die Auflösung des teilfixierten Gitters temporär um einen Level erhöht. Anschließend wird jedoch nicht wie im Vor- und Nachlauf der Wölbung bis Level 0, sondern nur bis Level 2 vergröbert. Das Beispiel zeigt zugleich die Funktionalität des Softwarepakets hinsichtlich einer gitteradaptiven Randbehandlung.

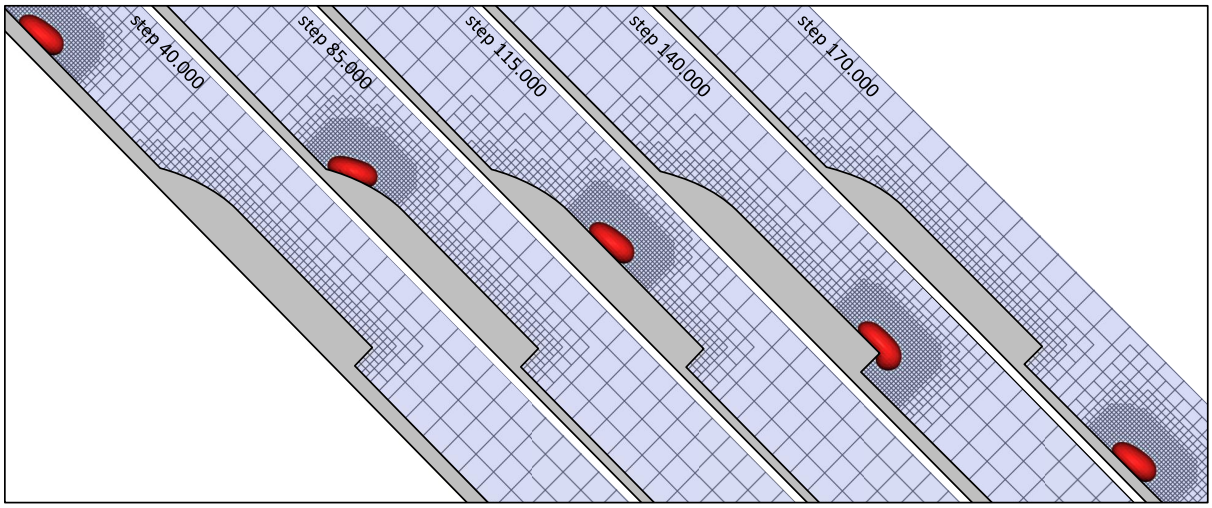


Abbildung 8.11: *Beispiel für ein adaptives, teilfixiertes Berechnungsgitter in 3-D (gleitender Tropfen, Schnitt)*

9 Validierung II (Mehrphase, seriell)

In diesem Kapitel wird das in Abschn. 4.3 vorgestellte erweiterte Rothmann-Keller-Modell mit Hilfe von VIRTUALFLUIDS verifiziert. Hierzu wird zunächst die Korrektheit der Oberflächenspannung für verschiedene Strömungsparameter bei Zweiphasensimulationen überprüft. Anhand der Zweiphasen-Spaltströmung folgt unter anderem eine Konvergenzstudie für Phasen unterschiedlicher Dichte. Simulationen aufsteigender Blasen, bei denen die dynamische Gitterstruktur zur Geltung kommt, vervollständigen die Validierung des Verfahrens und der Datenstruktur.

9.1 Oberflächenspannung

Die Überprüfung der Oberflächenspannung erfolgt mit der Young-Laplace-Gleichung:

$$\Delta p = 2\sigma\kappa \tag{9.1}$$

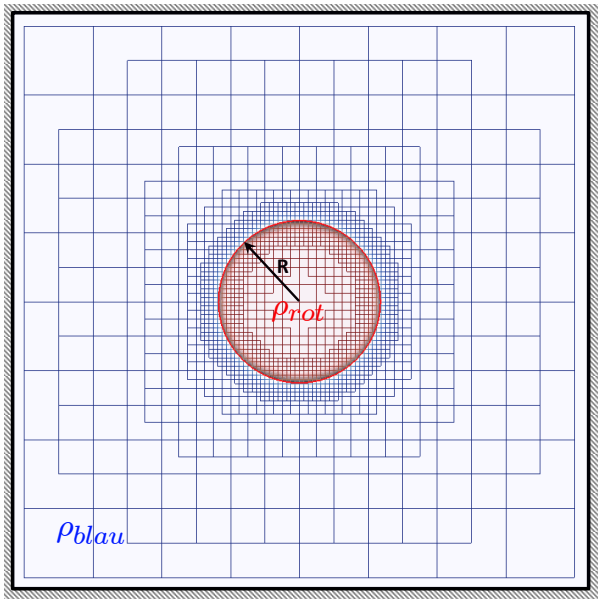
Sie beschreibt die Druckdifferenz Δp zwischen zwei statischen Fluiden in Abhängigkeit der Oberflächenkrümmung κ und -spannung σ. Für Kapillardrücke eines Flüssigkeitstropfens mit Radius R gilt für Mehrphasenprobleme mit kleiner Bond-Zahl (kleines Verhältnis von Gewichtskraft zur Kapillarkraft):

$$\Delta p = \frac{2\sigma}{R} \tag{9.2}$$

Die Berechnungen erfolgten auf einem 128x128x128 Gitter mit no-slip-Randbedingung, das für Simulationen auf nicht-uniformen Gittern anteilig vergröbert wurde (Abb. 9.1a). Die Ergebnisse für verschiedene Gitterkonfigurationen und Parameter stimmten sehr gut mit der analytischen Lösung überein (Abb. 9.1b). Simulationen mit nicht-uniformen Gittern, bei denen das Phaseninterface stets mit dem höchsten Level diskretisiert wurde, lieferten identische Ergebnisse.

Im direkten Vergleich zur Rechenzeit auf uniformen Gitter konvergierte das System um ein achtfaches schneller und benötigte eine Größenordnung weniger Rechenressourcen.

Die Druckverteilung innerhalb des Gebietes ist ebenfalls konsistent mit der Theorie (Abb. 9.2 und Abb. 9.3). Die Peaks beim Phasenübergang (Abb. 9.3a) sind Effekte des Entmischungsalgorithmus. Reduziert man den Entmischungsfaktor e (vgl. Abschn. 4.3) von $1,0$ auf $0,5$, so lassen sich diese Peaks bei minimaler Vergrößerung der Interfaceausdehnung und Beibehaltung der Genauigkeit deutlich verringern (Abb. 9.3b).

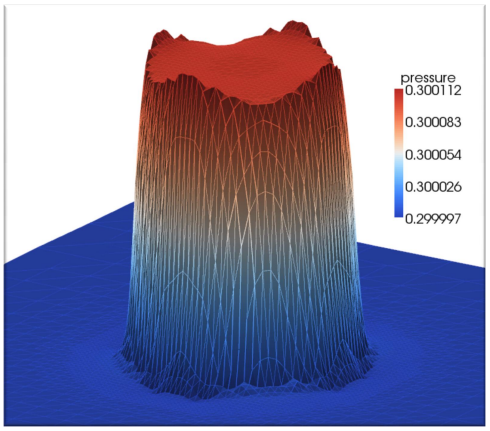


(a) Ruhende Blase in Box (128x128x128)

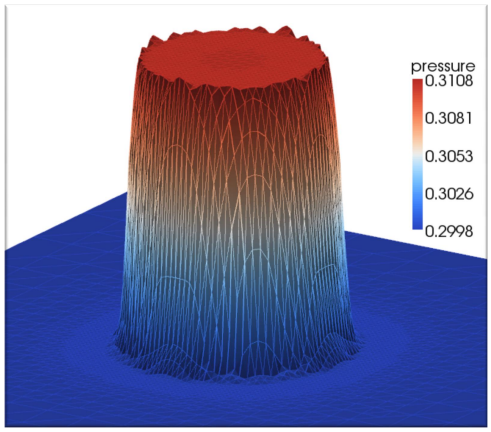
Fall	Parameter		Level	σ	Fehler [%]
$\rho_b = \rho_r$ $\nu_b = \nu_r$	ρ_b	1,0	4-4	0,001	0,0059
	T_b	1,0	0-4	0,001	0,0067
	ν_b	0,06	4-4	0,01	0,0059
	ρ_r	1,0	0-4	0,01	0,0054
	T_r	0,3	4-4	0,1	0,0079
	ν_r	0,06	0-4	0,1	0,0051
$\rho_b = \rho_r$ $\nu_b \neq \nu_r$	ρ_b	1,0	4-4	0,001	0,0027
	T_b	1/6	0-4	0,001	0,0043
	ν_b	0,1	4-4	0,01	0,0051
	ρ_r	1,0	0-4	0,01	0,0059
	T_r	0,1	4-4	0,1	0,0051
	ν_r	0,03	0-4	0,1	0,0043
$\rho_b \neq \rho_r$ $\nu_b = \nu_r$ mit Korr. (Gl. 4.22)	ρ_b	2,0	4-4	0,001	0,0059
	T_b	0,1	0-4	0,001	0,0059
	ν_b	0,5	4-4	0,01	0,0043
	ρ_r	0,06	0-4	0,01	0,0019
	T_r	0,4	4-4	0,1	0,0059
	ν_r	0,06	0-4	0,1	0,0051
$\rho_b \neq \rho_r$ $\nu_b = \nu_r$ ohne Korr. (Gl. 4.22)	ρ_b	2,0	4-4	0,001	0,0043
	T_b	0,1	0-4	0,001	0,0059
	ν_b	0,06	4-4	0,01	0,0035
	ρ_r	0,5	0-4	0,01	0,0043
	T_r	0,4	4-4	0,1	0,0035
	ν_r	0,06	0-4	0,1	0,0027

(b) Berechnungsergebnisse (r=rot, b=blau)

Abbildung 9.1: *Validierung der Oberflächenspannung*

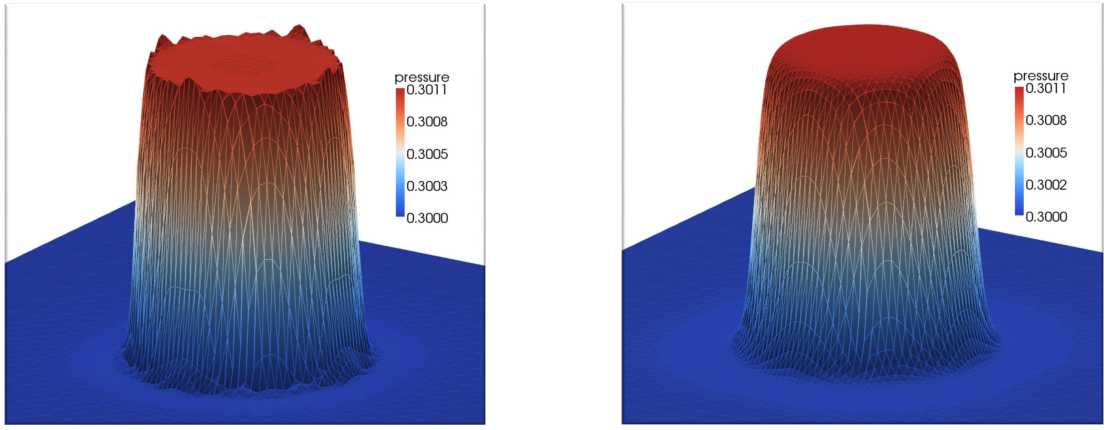


(a) $\sigma = 0,001$ *(500.000-fache Überhöhung)*



(b) $\sigma = 0,1$ *(5.000-fache Überhöhung)*

Abbildung 9.2: *Überhöhte Druckdarstellung für verschiedene Oberflächenspannungen*



(a) *e=1,0 (50.000-fache Überhöhung)* ***(b)*** *e=0,5 (50.000-fache Überhöhung)*

Abbildung 9.3: *Überhöhte Druckdarstellung für verschiedene Entmischungen ($\sigma = 0,01$)*

9.2 Spaltströmung zweier nicht-mischbarer Phasen

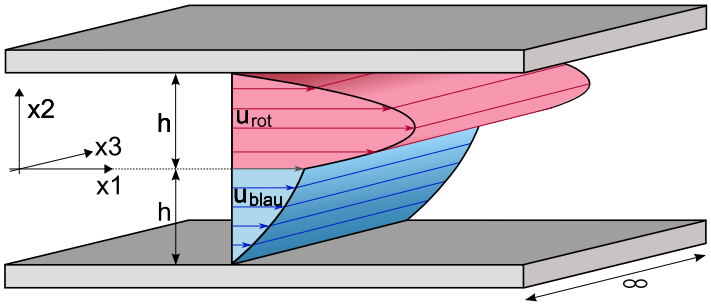


Abbildung 9.4: *Zweiphasenspaltströmung*

Hier bewegen sich, angetrieben von einem konstanten Druckgradient $\frac{dp}{dx_1}$, zwei nicht-mischbare, übereinander liegende Phasen in einem unendlich in x_1- und x_3-Richtung ausgedehnten Spalt mit einer zu den Kanalwänden parallelen, ebenen Grenzfläche. Im stationären Zustand stellt sich in Abhängigkeit der Parameter ein zu Abb. 9.4 ähnliches Geschwindigkeitsprofil ein. Die bereichsweise definierten Profile lassen sich, wie in [154] beschrieben, aus den Navier-Stokes-Gleichungen herleiten:

$$u_{rot} = \frac{dp}{dx_1}\left(\frac{1}{2\,\mu_{rot}}x_2^2 + \frac{h(\mu_{blau}-\mu_{rot})}{2\,\mu_{rot}\,(\mu_{blau}+\mu_{rot})}x_2 - \frac{h^2}{\mu_{blau}+\mu_{rot}}\right) \tag{9.3}$$

$$u_{blau} = \frac{dp}{dx_1}\left(\frac{1}{2\,\mu_{blau}}x_2^2 + \frac{h(\mu_{blau}-\mu_{rot})}{2\,\mu_{blau}(\mu_{blau}+\mu_{rot})}x_2 - \frac{h^2}{\mu_{blau}+\mu_{rot}}\right) \tag{9.4}$$

Die Simulationen wurden auf einem in x_1- und x_3-Richtung periodischen Gitter (62x10x10) durchgeführt. An den Kanalwänden wurde die no-slip-Randbedingung verwendet. Da die Oberflächenspannung aufgrund der ebenen Grenzfläche keine Auswirkungen hat, wurde sie zu Null gesetzt.

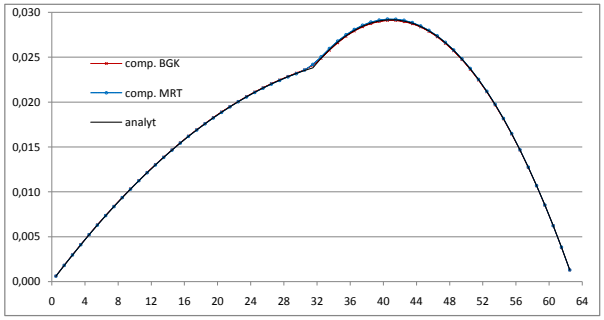
9.2.1 Simulation mit unterschiedlichen Viskositäten

Für die ersten Simulationen wurde für die Phasen jeweils die gleiche Dichte und unterschiedliche Viskosität verwendet (Tab. 9.1).

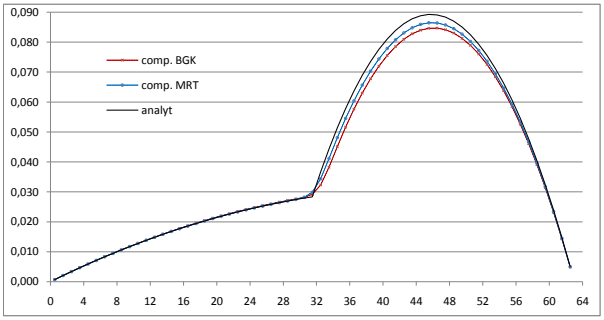
	ρ_{blau}	ρ_{rot}	τ_{blau}	τ_{rot}	μ_{blau}	μ_{rot}	$\frac{\mu_{blau}}{\mu_{rot}}$	$\frac{dp}{dx_1}$	h
Testfall A	1,0	1,0	3/2	0,75	1/3	1/12	4	0,00001	31,5
Testfall B	1,0	1,0	3/2	0,55	1/3	1/60	20	0,00001	31,5
Testfall C	1,0	1,0	3/2	0,51	1/3	1/300	100	0,000001	31,5

Tabelle 9.1: *Strömungsparameter für die ebene Zweiphasenspaltströmung*

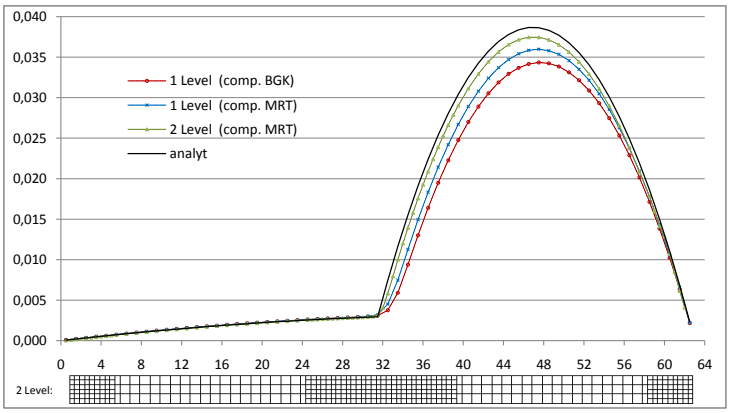
Für kleine Viskositätsverhältnisse entspricht das numerische dem analytischen Geschwindigkeitsprofil (Abb. 9.5a). Für große Unterschiede hingegen nehmen die Unterschiede sichtbar zu. Die Ursache ist die numerische Abbildung des wachsenden Phasenfeldgradienten (Abb. 9.5b). Wie in Abs. 4.3.2 erläutert, ist die Ausdehnung des Interfacebereiches abhängig von der jeweiligen Auflösung. Somit verschlechtert sich mit zunehmendem Gradienten dessen numerische Abbildung. Durch diese Abhängigkeit führt eine lokale Gitterverfeinerung im Phaseninterfacebereich zu genaueren Lösungen (Abb. 9.5c). Im direkten Vergleich der untersuchten LB-Modelle liefert das MRT-Modell gegenüber dem BGK-Ansatz die genaueren Ergebnisse.



(a) *Testfall A* $\left(\frac{\mu_{blau}}{\mu_{rot}} = 4\right)$



(b) *Testfall B* $\left(\frac{\mu_{blau}}{\mu_{rot}} = 20\right)$



(c) *Testfall C* $\left(\frac{\mu_{blau}}{\mu_{rot}} = 100\right)$

Abbildung 9.5: *Geschwindigkeitsprofile für Zweiphasenspaltströmung mit unterschiedlicher Viskosität und gleicher Dichte*

9.2.2 Simulation mit unterschiedlichen Dichten

Verwendet man anstelle von unterschiedlichen Viskositäten unterschiedliche Dichten, so erhält man unter Vernachlässigung der in Abschn. 4.3 vorgestellten viskosen Korrekturterme ein fehlerhaftes Geschwindigkeitsprofil (Abb. 9.6). Dieser Effekt wurde bereits in [51] erwähnt. Die Verwendung der Korrekturterme führt zu einer deutlichen Verbesserung des Ergebnisses. Eine geringfügige Störung im Übergangsbereich bleibt jedoch, auch bei Verwendung unterschiedlicher Diskretisierungssterne und Interpolationsverfahren, bestehen.

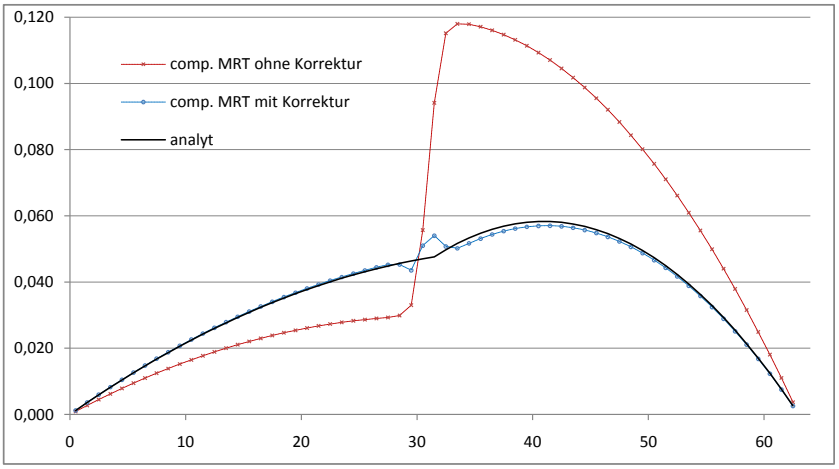


Abbildung 9.6: *Geschwindigkeitsprofile für Zweiphasenspaltströmung mit gleicher Viskosität und unterschiedlichen Dichten* $\left(\frac{\rho_{blau}}{\rho_{rot}} = 4\right)$

Um die Konvergenz des Verfahrens für unterschiedliche Dichten zu ermitteln, wurde ein Kanal mit $h = 50$ und den Dichten $\rho_{blau} = 4$ und $\rho_{rot} = 1$ für verschiedene Diskretisierungen sowie verschiedene Entmischungsfaktoren e berechnet (Tab. 9.2). Die Fehlerberechnung berücksichtigt jeden diskreten Berechnungspunkt und basiert auf der diskreten L1 Norm:

$$E1 = \frac{\sum |u_{num} - u_{analyt}|}{\sum |u_{analyt}|} \tag{9.5}$$

H (Knoten)	Δx	ν	F_{blau}	F_{rot}	E1 [%]		
					e=1,0	e=0,5	e= 0,25
10	10,000	1,7000	1,00E-04	4,00E-04	5,77	9,99	20,68
20	5,000	0,8500	2,50E-05	1,00E-04	3,23	3,87	8,68
25	4,000	0,6800	1,60E-05	6,40E-05	2,06	3,05	6,85
40	2,500	0,4250	6,25E-06	2,50E-05	2,73	1,00	2,55
50	2,000	0,3400	4,00E-06	1,60E-05	2,72	0,86	1,65
80	1,250	0,2125	1,56E-06	6,25E-06	2,53	0,82	0,60
100	1,000	0,1700	1,00E-06	4,00E-06	2,49	0,75	0,34
125	0,800	0,1360	6,40E-07	2,56E-06	2,20	0,70	0,33
160	0,625	0,1063	3,91E-07	1,56E-06	2,46	0,78	0,23
200	0,500	0,0850	2,50E-07	1,00E-06	2,43	0,79	0,19

Tabelle 9.2: *Parameter und E1-Fehler für verschiedene Auflösungen und Entmischungsfaktoren e* $\left(\frac{\rho_{blau}}{\rho_{rot}} = 4\right)$

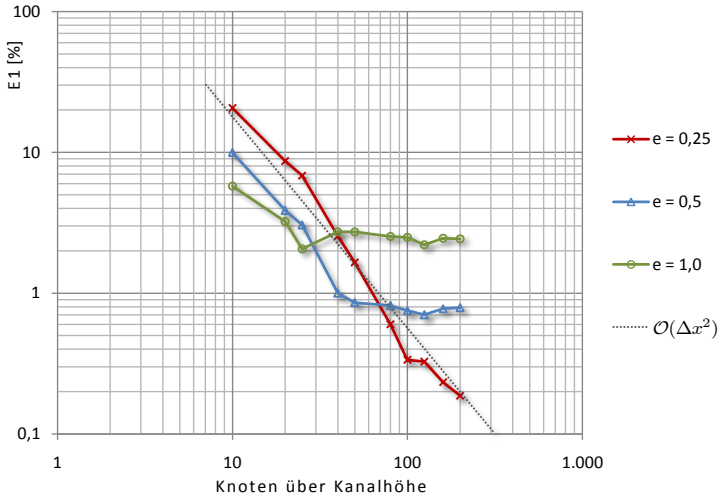
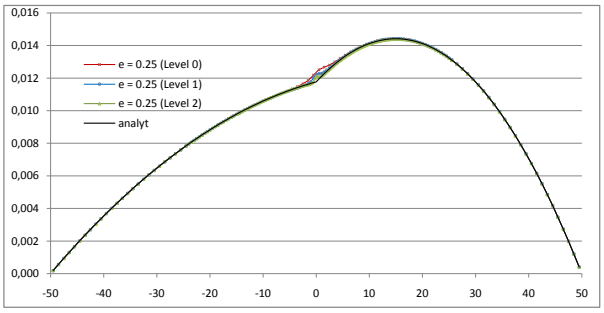


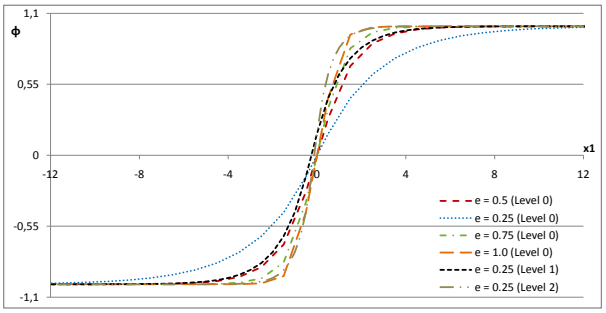
Abbildung 9.7: *Logarithmischer E1-Plot für $e = 1,0$, $e = 0,5$ und $e = 0,25$*

Wie in Abb. 9.7 dargestellt, konvergiert das Verfahren unter Verwendung eines kleinen Entmischungsfaktors und Berücksichtigung der Korrekturterme mit quadratischer Konvergenzordnung im Raum. Bei diesem wird durch die größere Ausdehnung des Phaseninterfaces der Einfluss der Störung reduziert (vgl. Abb. 9.8).

Um die Auswirkung lokaler Verfeinerung zu testen, wurde der Testfall mit 100 Knoten über die Kanalhöhe für $e = 0,25$ im Phasenübergangsbereich um einen bzw. zwei Level verfeinert. Wie in Abb. 9.9a zu sehen, reduziert sich sowohl die Amplitude als auch die Ausdehnung der Störung mit jeder Verfeinerungsstufe. Zudem wird die physikalische Ausdehnung des Phaseninterfaces reduziert. So entspricht die Breite in Level 2 bei einem Entmischungsfaktor von 0,25 ungefähr der einer Entmischung mit $e = 1,0$ im Level 0 (Abb. 9.9b).



(a) *Geschwindigkeitsprofil bei lokaler Verfeinerung*



(b) *Phaseninterfaceverlauf*

Abbildung 9.9: *Lokale Verfeinerung für H=100 und e=0,25*

Bei der Berechnung des E1-Fehlers werden die Knoten des nicht-uniformen Netzes entsprechend ihres Einzugbereichs gewichtet. Wie erwartet, reduziert sich der Fehler mit der Verfeinerung:

Knoten	Δx	ν	F_{blau}	F_{rot}	Level 0	Level 1	Level 2
100	1,0	0,17	1,0E-06	4,0E-06	0,0034	0,0013	0,00081

Tabelle 9.3: *E1-Fehler bei lokaler Verfeinerung ($e = 0,25$)*

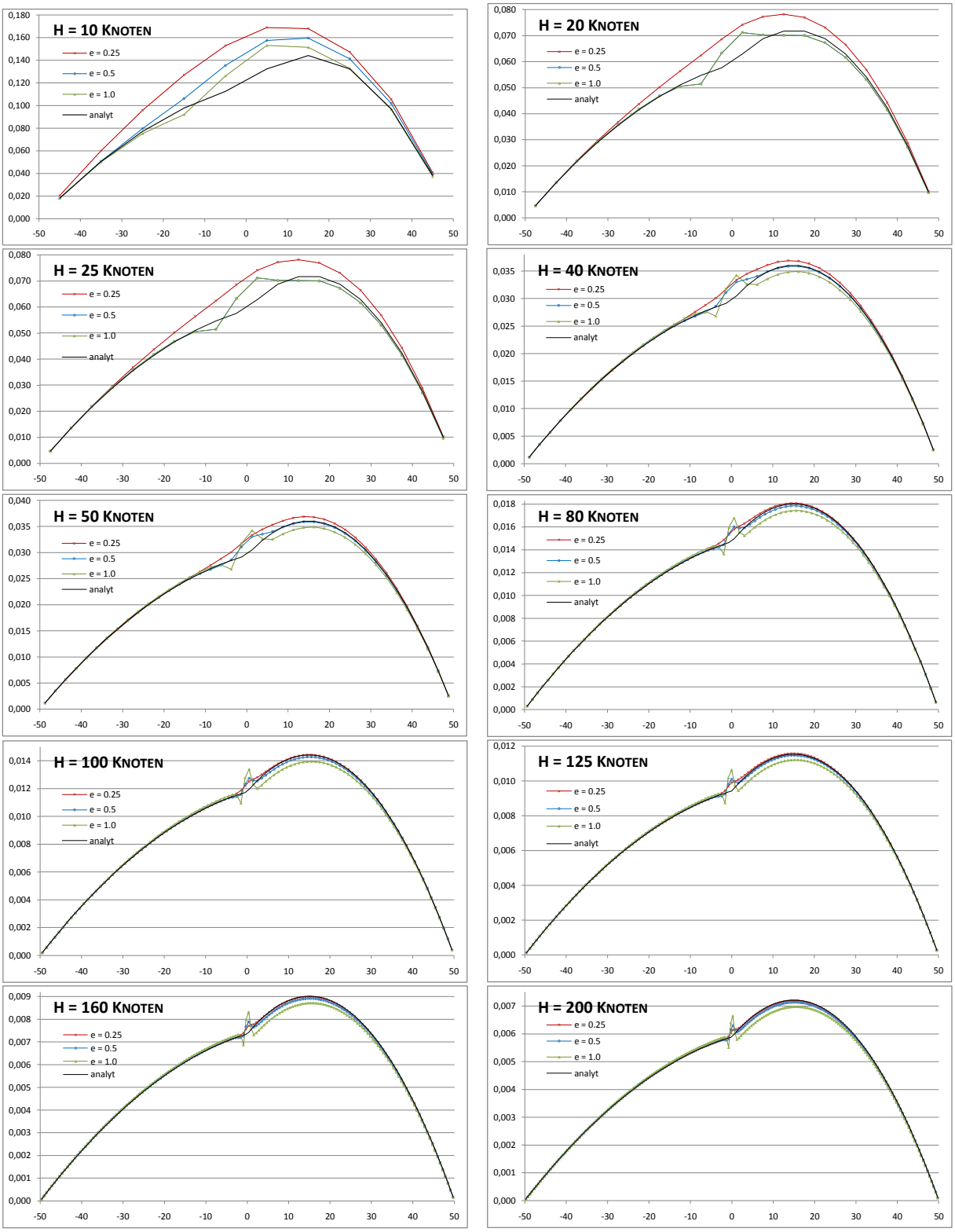


Abbildung 9.8: *Geschwindigkeitsprofile für unterschiedliche Auflösungen und Entmischungsfaktoren* $\left(\frac{\rho_{blau}}{\rho_{rot}} = 4\right)$

9.3 Aufsteigende Blasen

Aufsteigende Blasen eignen sich aufgrund des sich permanent örtlich ändernden Phaseninterfaces unter anderem für die Validierung der in Kap. 8 besprochenen Gitteradaptivität. Wie in [154] beschrieben, kann mittels Dimensionsanalyse gezeigt werden, dass eine Blase in einem unendlich ausgedehnten Medium unter Einwirkung der Schwerkraft g durch vier dimensionslose Kennzahlen bestimmt werden kann:

- *Reynoldszahl:*
$$Re = \frac{u\,d}{\nu} \tag{9.6}$$
- *Eötvös-Zahl:*
$$Eo = \frac{g\,\Delta\rho\,d^2}{\sigma} \tag{9.7}$$
- *Morton-Zahl:*
$$M = \frac{g\,\mu^4\,\Delta\rho\,d^2}{\rho^2\,\sigma^3} \tag{9.8}$$
- *Viskositätsverhältnis:*
$$\gamma = \frac{\mu_{Blase}}{\mu} = \frac{\nu_{Blase}\,\rho_{Blase}}{\nu\,\rho} \tag{9.9}$$

Hierbei ist u die Aufstiegsgeschwindigkeit der Blase und d der zu ihrem Volumen äquivalente Kugeldurchmesser. Mittels dieser Kenngrößen kann mit Hilfe von Abb. 9.10 die Form der jeweiligen Blase abgeschätzt werden. Die Simulationen erfolgten für das sphärische und das Kugelkappenregime. Die

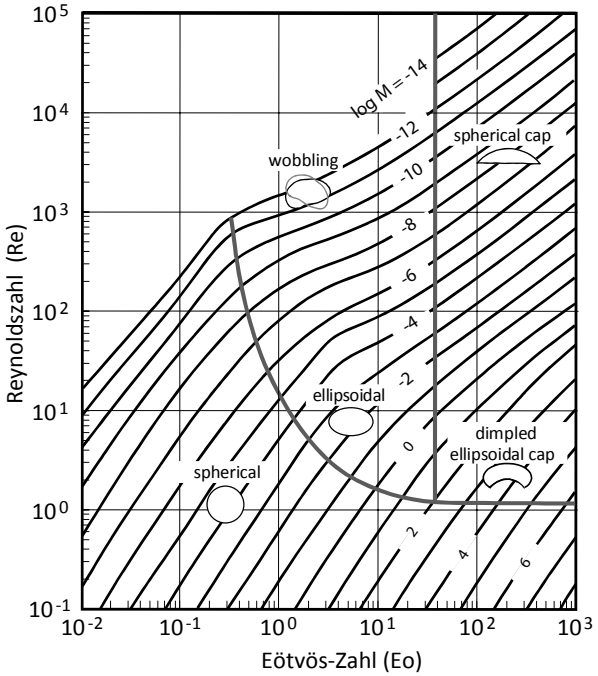


Abbildung 9.10: *Oberflächenregimes für Blasen und Tropfen (aus [19])*

Blasen stiegen jeweils in einem Zylinder mit dem Durchmesser D auf. Die analytischen Berechnungsgrundlagen wurden aus [154] übernommen. Für Berechnungen im sphärische Regime mit $Re < 1$ in

zylindrischen Gefäßen mit $\lambda = \frac{d}{D}$ kann somit die semianalytische Lösung zur Berechnung der Aufstiegsgeschwindigkeit verwendet werden:

$$u_{Zyl} = \frac{1}{6}\frac{g d^2 \Delta\rho}{\mu}\frac{3\gamma+1}{\gamma+2}\frac{1}{K} \tag{9.10}$$

mit

$$K = \frac{1 + 2{,}2757\,\lambda^5 \frac{1-\gamma}{2+3\gamma}}{1 - 0{,}7017\,\frac{2+3\gamma}{1+\gamma}\lambda + 2{,}0865\,\frac{\gamma}{1+\gamma}\lambda^3 + 0{,}5689\,\frac{2-3\gamma}{1+\gamma}\lambda^5 - 0{,}72603\,\frac{1-\gamma}{1+\gamma}\lambda^6} \tag{9.11}$$

Im Kugelkappenregime kann für $Eo > 40$ und $M > 200$ die Aufstiegsgeschwindigkeit näherungsweise aus

$$2Re^2 + 6Re\frac{2+3\gamma}{1+\gamma} - \frac{Eo^{\frac{2}{3}}}{\sqrt{M}} = 0 \tag{9.12}$$

ermittelt werden:

$$u = \nu\, d\, \frac{(-6-9\gamma)\sqrt{M} \pm \sqrt{\left(36+108\gamma+81\gamma^2\right)M + \left(2+4\gamma+2\gamma^2\right)\sqrt{M}Eo^{\frac{2}{3}}}}{(2+2\gamma)\sqrt{M}} \tag{9.13}$$

Die Rohrwände werden dabei für $\lambda < 0{,}6$ mit

$$u_{Zyl} = 1{,}13\, e^{-\lambda} \tag{9.14}$$

berücksichtigt.

$\#x_1$ $\#x_2$ $\#x_3$	ρ; ρ_B	T; T_B	ν; ν_B	D d	σ g	Re Eo M	Level	Δx_f	Korr.	u_{analyt}	u_{num}	rel. Fehler [%]
128	2,2	0,1	1/6	127	0,02	0,53	0-3	1	-	0,003	0,0046	53,33
128	0,55	0,4	1/6	30	1,5E-5	1,11	0-3	1	+		0,0032	6,67
512						0,0116	3-3	1	+		0,0031	5,90
64	2,37	0,12	1/6	63	0,05	0,51	0-4	1	+	0,0043	0,0045	4,65
64	0,59	0,48	1/6	20	6,0E-5	0,856	0-5	0,5	+		0,0046	6,97
416						0,0037						
128	2,02	0,2	1/30	127	0,05	0,34	0-3	1	-	7,6E-4	0,0012	57,9
128	1,01	0,4	1/3	15	5,0E-6	0,379	0-3		+		7,1E-4	6,57
512						9,4E-4	3-3		+		7,2E-4	5,26

Tabelle 9.4: *Ergebnisse für Blasen im sphärischen Regime*

Die Ergebnisse (Tab. 9.4, 9.5) stimmen bei Verwendung der Korrekturterme gut mit den Näherungslösungen überein. Auch die Geometrie der Blase entspricht dem jeweiligen Regime. Bei $\gamma \to \infty$ ist das Strömungsfeld innerhalb der Blase nur schwach ausgeprägt (Abb. 9.12a) und die Aufstiegsgeschwindigkeit geringer als bei $\gamma \to 0$, bei denen im Inneren der Blase gut die auftretenden Hill-Wirbel zu erkennen sind (Abb. 9.12b). In Abb. 9.11 ist beispielhaft der Druckunterschied im Inneren einer aufsteigenden Blase im Verhältnis zum Umgebungsdruck abgebildet.

Wie erwartet liefern Simulationen ohne Verwendung der Korrekturterme unzureichende Ergebnisse. Berechnungen mit adaptiver Gitterstruktur erzielten gleichwertige Ergebnisse zu denen mit uniformen Gittern. Jedoch wurden die Simulationszeit sowie der Speicherbedarf durch die Minimierung der Freiheitsgrade erheblich reduziert. Eine Berechnung der Zylindertestfälle 2048x2048x416 würde

uniform jeweils ungefähr $2,5 \cdot 10^{10}$ Freiheitsgrade benötigen. Unter Verwendung des acht Byte großen Datentyps double pro Partikelverteilung f_i würde dies einem Speicherbedarf von 186 GB allein für die Strömungsinformation entsprechen (ohne f_{temp}-Feld). Mit einer solchen Datenmenge sind kleinere Computercluster oftmals schon überfordert. Die adaptive Berechnung benötigte im Schnitt zirka $3 \cdot 10^6$ Freiheitsgerade und konnte somit auf einem handelsüblichen PC innerhalb weniger Stunden durchgeführt werden (Abb. 9.13).

$\#x_1$ $\#x_2$ $\#x_3$	$\rho_;$ ρ_B	$T_;$ T_B	$\nu_;$ ν_B	D d	σ g	Re Eo M	Level	Δx_f	Korr.	u_{analyt}	u_{num}	rel. Fehler [%]
64	8,51	0,07	0,15	63	0,001	0,7	0-5	1	-	0,0052	0,00805	54,8
64	1,20	0,49	0,10	20	4,0E-5	116	0-5	1	+		0,00508	2,31
416						10682	0-5	1	+		0,00508	2,31
							1-5	1	+		0,00493	5,19
							2-5	1	-		0,00766	47,3
							2-5	1	+		0,00484	6,92
							4-5	1	-		0,00745	43,3
							4-5	1	+		0,00479	7,88
							5-5	1	+		0,00480	7,69
128	8,38	0,07	0,215	127	0,005	2,63	0-5	1	-	0,00117	0,00163	39,3
128	1,20	0,49	0,215	40	4,0E-5	92	0-5	1	+		0,00108	7,69
384						344	0-5	1	+		0,00107	8,55
							0-6	0,5	+		0,00107	8,55
							0-6	0,5	+		0,00106	9,40
2048	8,298	0,07	0,215	2047	0,005	3,62	0-5	1	-	0,00156	0,00221	41,1
2048	1,185	0,49	0,215	40	4,0E-5	91	0-5	1	+		0,00145	7,05
384						335						
2048	8,466	0,07	0,215	2047	0,005	2,13	0-5	1	+	0,0013	0,00128	1,54
2048	1,209	0,49	0,215	35	4,0E-5	71						
416						356						
2048	8,466	0,07	0,166	2047	0,001	3,18	0-5	1	+	0,0015	0,00142	5,33
2048	1,209	0,49	0,166	35	4,0E-5	356						
416						16052						

Tabelle 9.5: *Ergebnisse für Blasen im Kugelkappenregime*

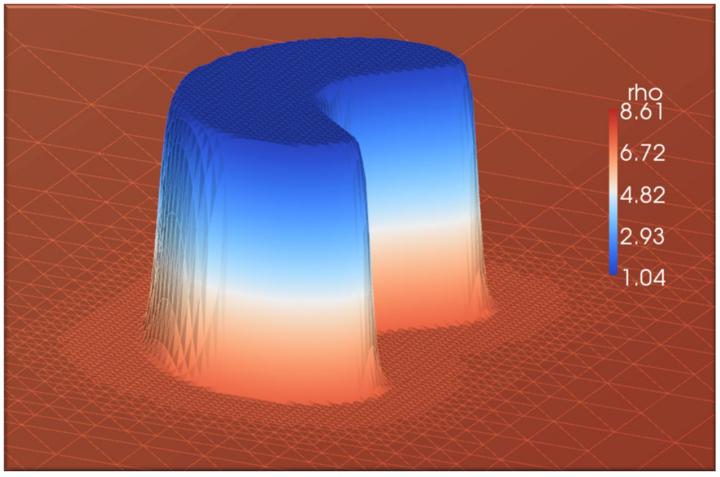
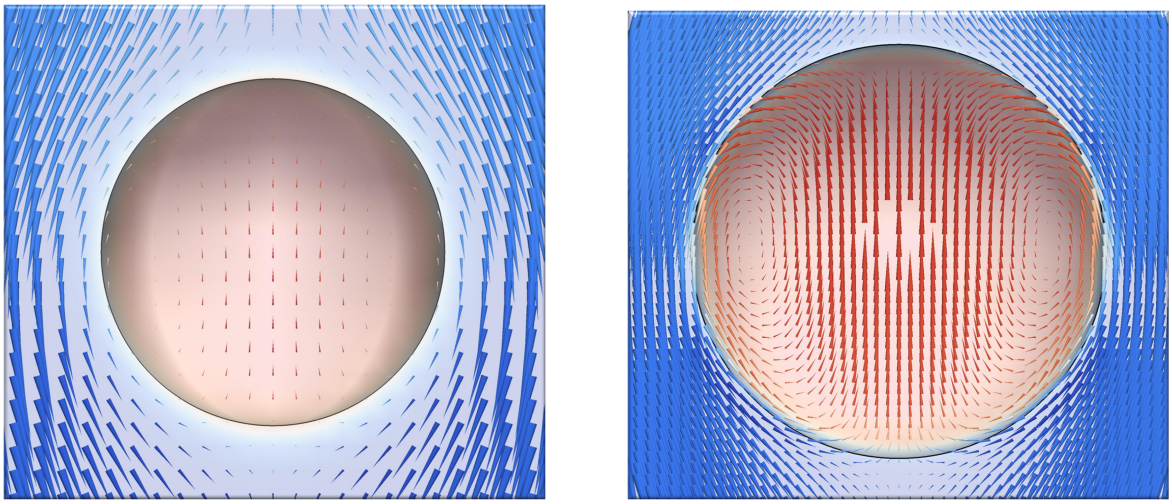


Abbildung 9.11: *Überhöhte Darstellung der Dichte in Blasenmitte*



(a) *Fest* ***(b)*** *Flüssig*

Abbildung 9.12: *Strömungsfelder von Blasen im sphärischen Regime (Schnitt durch Blasenmitte)*

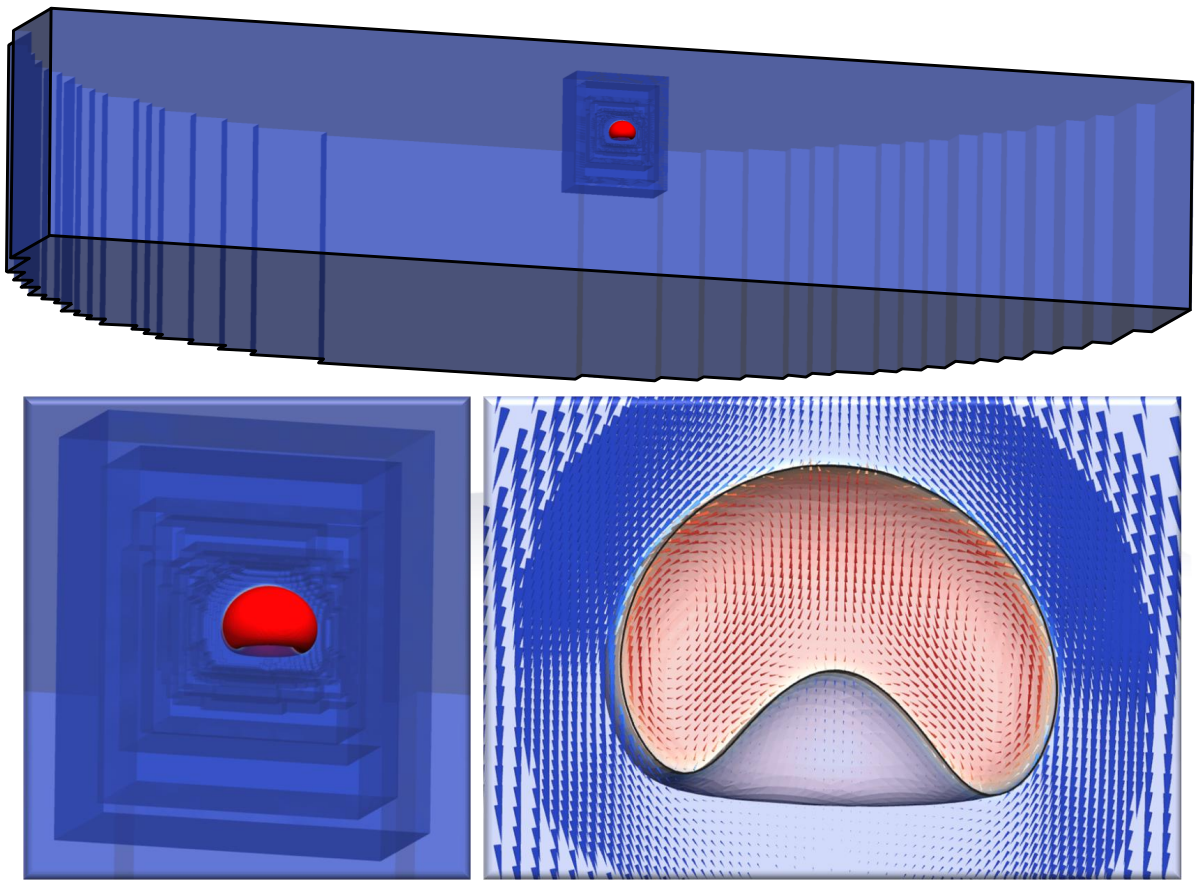


Abbildung 9.13: *Schnitt durch Zylindermitte, Isoflächen der adaptiven Gitterlevel und Blase (2048x2048x416)*

10 Erweiterung II - verteilter Berechnungskern

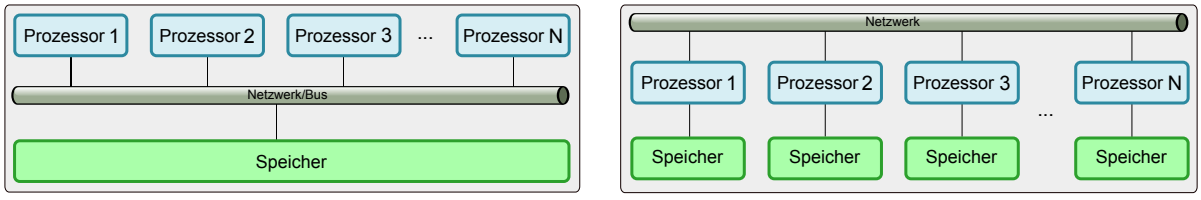
Seit den 1970er Jahren nimmt die Zahl der Großrechner kontinuierlich zu. Zu Beginn waren diese sehr kostspielig und oftmals aufwändig zu programmieren. Die Bedingungen verbesserten sich mit dem Aufkommen kostengünstiger PC-Cluster, die mit Unix und Windows Compute Cluster auch nicht-proprietäre Betriebssysteme mit einer Vielzahl standardisierter Entwicklungswerkzeugen und -bibliotheken zur Verfügung stellten. Dadurch war es erstmals möglich, auf breiter Basis verteilt rechnende Software zu erstellen.

In der Literatur ist oft zu lesen, dass sich das Lattice-Boltzmann-Verfahren aufgrund seiner regulären Gitter sehr gut zur Parallelisierung eignet [28, 88, 114, 118, 119, 131, 136]. Diese Aussage trifft im Allgemeinen nur auf Implementierungen für uniforme Gitter mit lokalen Randbedingungen, wie dem Simple-Bounce-Back, zu. Bei Gitterverfeinerung, Randbedingungen höherer Ordnung und Mehrphasenproblemen besteht jedoch die Notwendigkeit nicht-lokaler Berechnungsschritte, die eine Parallelisierung nachhaltig erschwert.

Es wird nun ein Parallelisierungsansatz in 2-D auf Basis eines abstrakten Regelwerks für VIRTUALFLUIDS vorgestellt, das vom Benutzer definierte Diskretisierungssterne berücksichtigt. Die Gebietszerlegung erfolgt dabei unter Zuhilfenahme der METIS-Bibliothek. Für die Interprozesskommunikation kommt das Message Passing Interface (MPI) zum Einsatz. Eine Umsetzung des Konzepts für den 3-D-Knotencode ist grundsätzlich möglich. In dieser Arbeit wurde auf diese jedoch zugunsten der Entwicklung der in Teil III besprochenen hybriden Blockgittererweiterung von VIRTUALFLUIDS verzichtet.

10.1 Speicherarchitekturen

Bei der Parallelisierung ist die Speicherarchitektur des Zielsystems für das zu verwendende Design ausschlaggebend.



(a) *Shared-Memory* ***(b)*** *Distributed-Memory*

Abbildung 10.1: *Speicherarchitekturen*

Im Wesentlichen unterscheidet man hier zwischen zwei verschiedenen Architekturen. Auf der einen Seite gibt es die sogenannte **Shared Memory Architecture (SMA)**, bei der sich die Rechenkerne einen

gemeinsamen Arbeitsspeicher teilen (Abb. 10.1a). Zum Einsatz kommt diese vorzugsweise bei Rechnern mit Mehrkernprozessoren. Alle Kerne greifen, meist über denselben Bus, auf den Systemspeicher zu. Intern realisieren die meisten Programme den parallelen Ablauf mit Hilfe von mehreren Threads. Bei der Implementierung besteht die Schwierigkeit diese Zugriffe zu synchronisieren, um z. B. Race-Conditions zu vermeiden.

Mit vergleichsweise wenig Aufwand kann eine Codeparallelisierung mit OpenMP [115] erreicht werden. Die Parallelisierung wird dabei halbautomatisch mit Hilfe von Präprozessordirektiven erzielt. OpenMP ist für eine Vielzahl von Programmiersprachen, wie z. B. C, C++ und Fortran verfügbar. Das Ergebnis einer OpenMP-Parallelisierung hängt von vielen Faktoren, wie z. B. der verwendeten OpenMP-Implementierung oder der im System vorhandenen Speicherbandbreite, ab.

In [163] sind diverse algorithmische Optimierungsvorschläge hinsichtlich LB-Implementierungen auf Mehrkernprozessoren aufgeführt. Zudem wird dort ein Codegenerator vorgestellt, der automatisch optimierten LB-Quellcode für unterschiedliche Plattformen, wie z. B. Intel^{TM} Clovertown, AMD Opteron^{TM} X2, Sun Niagara2 und STI Cell, erstellt. Für die Parallelisierung wird die POSIX-Threads-Bibliothek verwendet.

Der SMA steht die **verteilte** (*engl.: distributed*) **Speicherarchitektur** gegenüber. Diese kommt vor allem in PC-Clustern und Großrechnern zum Einsatz. Jeder Kern hat seinen eigenen Arbeitsspeicher. Zugriffe untereinander erfolgen über Interprozessschnittstellen, wobei die Daten z. B. mit Hilfe eines schnellen Netzwerkes übertragen werden (Abb. 10.1b). Das Erstellen einer verteilten Anwendung für einfache Probleme erfordert einen höheren Aufwand als eine im Vergleich einfache OpenMP-Parallelisierung. Dieser Mehraufwand amortisiert sich in der Regel bei komplexen Problemen und großen Systemen, für die eine optimale OpenMP-Parallelisierung nicht trivial ist, durch eine höhere parallele Effizienz. Zudem arbeiten für verteilte Systeme entwickelte Programme meist problemlos auch auf SMA-Systemen, was umgekehrt nicht möglich ist.

In modernen Großrechnern sind heutzutage beide Systeme zu finden. Ein Knoten eines solchen Rechners besitzt meist mehrere Mehrkernprozessoren, die sich einen gemeinsamen Speicher teilen. Die Kommunikation zwischen den einzelnen Knoten erfolgt dann über ein Netzwerk. Auf solchen Rechnern existiert oft eine virtuelle Shared-Memory-Umgebung [123], die es dem Programmierer ermöglicht, den gesamten, physikalisch verteilten Speicher als Shared-Memory zu nutzen. Der durch die Emulation verbundene Mehraufwand kann sich jedoch je nach Anwendung negativ auf die Leistung auswirken. Sinnvoll ist hier ein Hybridansatz, bei dem z. B. auf den einzelnen Knoten eine Parallelisierung mittels Threads umgesetzt wird und zwischen den einzelnen Knoten mit Hilfe von MPI Daten ausgetauscht werden.

10.2 Programmablauf

Die hier besprochene parallele Erweiterung von VIRTUALFLUIDS wurde in Hinblick auf PC-Clustersysteme mit verteiltem Speicher entworfen und implementiert. Um die Parallelisierung allgemeingültig zu halten und sie später allen Berechnungskernen zur Verfügung zu stellen, wird mit der Parallelsierungsschicht eine neue Abstraktionschicht zwischen der Topologie- und der Gitterschicht eingeführt (Abb. 10.2). Dadurch sind später für die parallele Erweiterung eines Berechnungskerns der Physikschicht nur minimale Ergänzungen notwendig. Der Programmierer muss sowohl das spezielle Knotengitter als auch den zugehörigen Knotendeskriptor von der zugehörigen parallelen Basisklasse ableiten. Zusätzlich müssen einige abstrakte Methoden des parallelen Gitters implementiert werden. Diese beschränken sich auf das Füllen und Verteilen der zu sendenden bzw. zu verteilenden Datenvektoren zwischen den individuellen Berechnungsprozessen (Abb. 10.3).

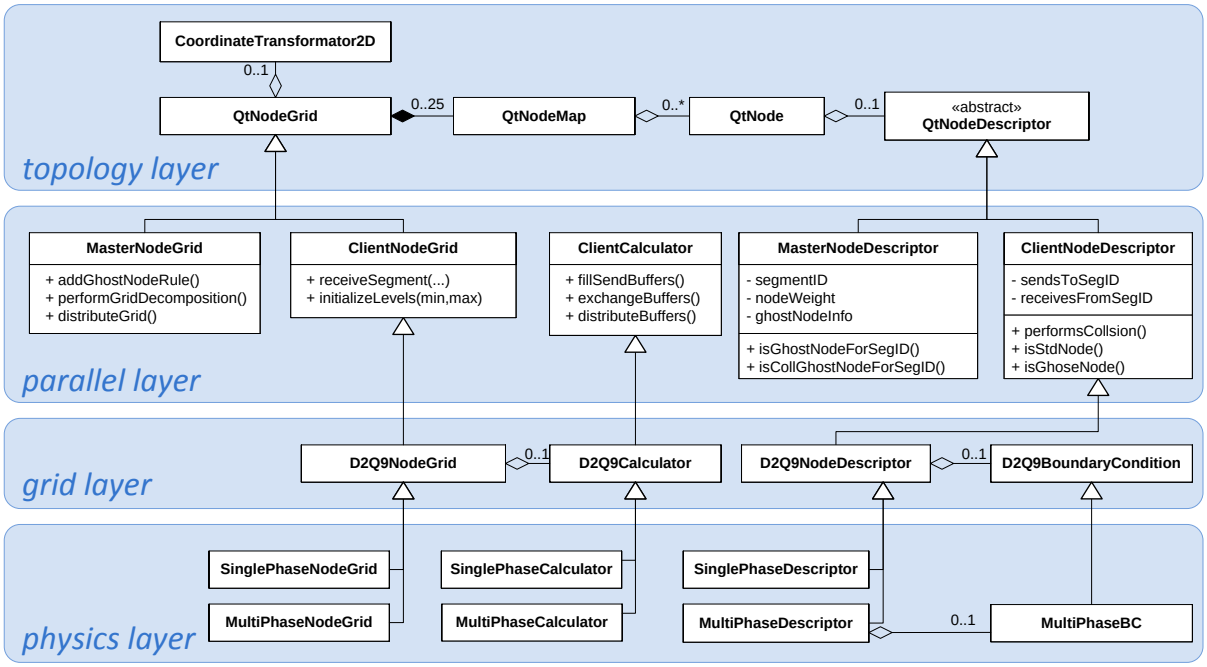


Abbildung 10.2: *Paralleles Knotengitter (UML)*

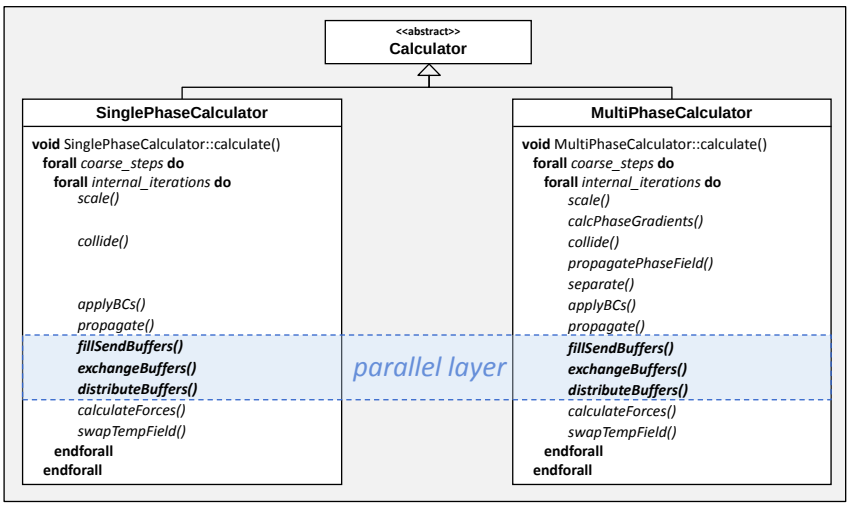


Abbildung 10.3: *Erweiterte Berechnungsroutine für verteilte Simulationen (Ein- und Mehrphase)*

Auch wenn die Daten innerhalb der Austauschroutine nicht blockierend versendet werden, handelt es sich global um einen synchronisierten Austauschprozess. Eine asynchrone Kommunikation wurde hinsichtlich der erhöhten Komplexität der Implementierung seitens der Berechnungskerne und der möglichen Nebeneffekte bei der Behandlung nicht-lokaler Diskretisierungssterne verworfen.

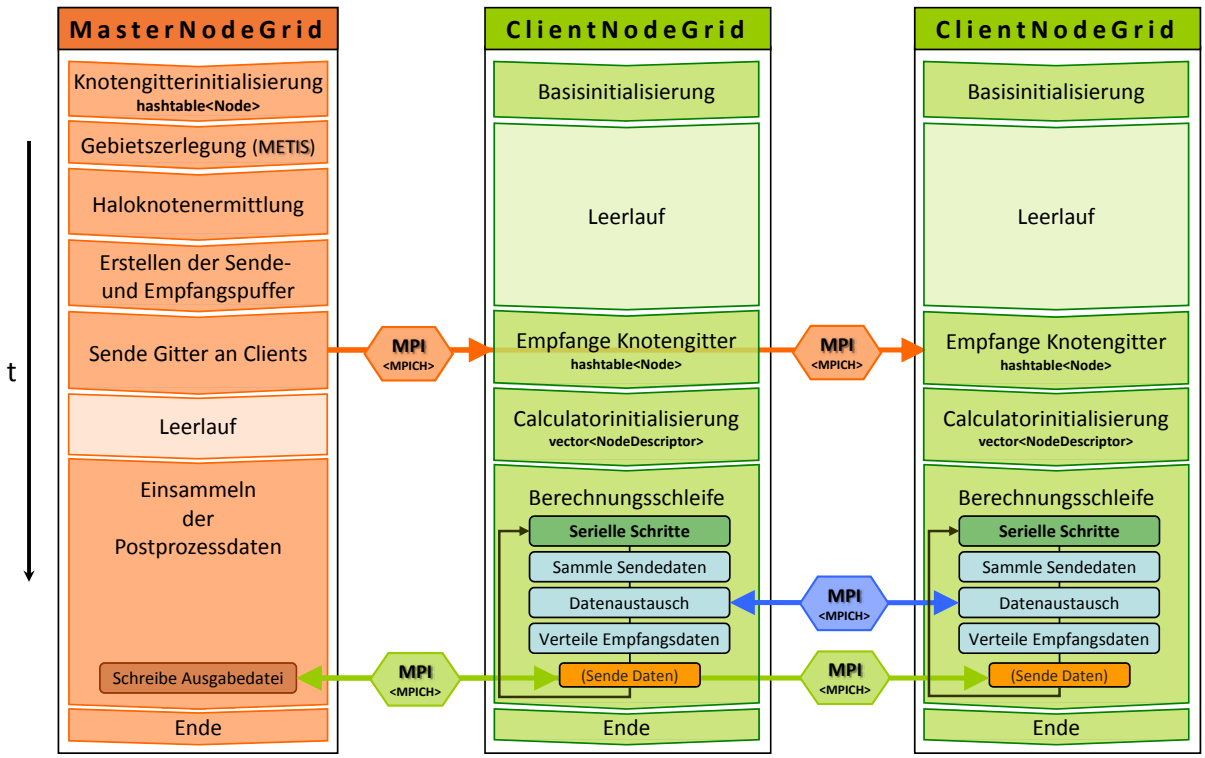


Abbildung 10.4: *Ablaufdiagramm einer verteilten Simulation mit* VIRTUALFLUIDS

Das verwendete Konzept sieht ein Hauptgitter (MasterNodeGrid) vor, das die gesamte Knotentopologie vorhält und keine Physikinformationen speichert. Auf diesem Gitter wird u. a. die Gebietszerlegung durchgeführt. Die ermittelten Teilgebiete werden zu den entsprechenden ClientNodeGrids gesendet, die anschließend die speziellen Knotengitter inklusive der Physikdaten, wie Viskosität, Randbedingungen und die LB-Verteilungen beinhaltenden Deskriptoren, initialisieren. Abb. 10.4 zeigt den Ablauf einer typischen verteilten Simulation mit VIRTUALFLUIDS.

Bei der Interprozesskommunikation wird zwischen drei verschiedenen Typen unterschieden:

- Master → Client (z. B. Übertragung der Teilgebiete)
- Client → Master (z. B. Einsammeln und Verarbeiten der Postprozessdaten)
- Client → Client (z. B. Austausch der Gebietsränder während der Berechnung)

Letztere bestimmt maßgeblich die parallele Effizienz des Lösers. Die Interprozesskommunikation basiert auf dem hocheffizienten Message Passing Interface (MPI) [134]. Hierfür existiert eine Vielzahl kommerzieller und nicht kommerzieller Implementierungen (MPICH [61], OpenMPI [42], Intel MPI [82], etc.) für verschiedenste Netzwerkschnittstellen (TCP, Myrinet [108], etc.).

10.3 Gebietszerlegung

Im Präprozess einer parallelen Simulation wird zunächst die gesamte Topologie auf dem MasterGrid initialisiert. Hierfür werden für die Gitterverfeinerung und die Bestimmung der Fluid- und Randknoten die entsprechenden Module des seriellen Lösers verwendet. Die Qualität der nachfolgenden Gebietszerlegung ist entscheidend für die spätere parallele Leistungsfähigkeit des Lösers. Hier gilt es die Knoten hinsichtlich einer optimalen Lastverteilung (*engl.: load balancing*) auf die Rechenclients zu verteilen. Insbesondere die unterschiedlichen Iterationen infolge des gestaffelten Zeitschrittverfahrens sowie der durch die Skalierungen erhöhte Rechenaufwand im Gitterinterface sind zu berücksichtigen.

Hierfür bietet sich die nicht-kommerzielle METIS-Bibliothek [89] an, die zur Partitionierung von irregulären Graphen und Finite-Element-Netzen verwendet werden kann. Die baumbasierte Datenstruktur wird für diesen Zweck in die spezielle METIS-Graphenstruktur konvertiert. Jeder Fluidknoten repräsentiert dabei einen Knoten des Graphen. Die diskrete Kommunikationsrichtung $\mathbf{e}_i$ zum Nachbarknoten beschreibt eine Kante innerhalb des Graphen. Die Wahl der spezifischen Knoten- und Kantenwichtungen unterliegt einer heuristischen Optimierung und kann deshalb vom Anwender selbst durchgeführt werden.

Standardmäßig werden die levelspezifischen Iterationen mit einem Knotengewicht von $2^{Knotenlevel\text{-}Basislevel}$ berücksichtigt, wodurch die maximale Anzahl der Gitterlevel innerhalb eines Systems beschränkt wird. Bei komplexen Graphen mit mehr als acht Gitterleveln kommt es zu nicht-deterministischen Abstürzen der in ANSI-C geschriebenen METIS-Bibliothek. Dieses Verhalten steht möglicherweise im Zusammenhang mit einer Überschreitung des integer-Wertelimits, mit der die Wichtungen innerhalb METIS 4.0 repräsentiert werden. Zukünftige METIS-Versionen (≥ 5) sollen Fließkommazahlen für die Wichtungen unterstützen.

Die Knotenkommunikation innerhalb eines Levels ist unabhängig vom Verfeinerungsgrad eines Knotens. Den entsprechenden Kanten wird deshalb ein Gewicht von eins zugewiesen. Aufgrund der zeitaufwändigen Skalierungsroutinen im Gitterübergangsbereich werden die Kantenwichtungen dort gemäß Abb. 10.5 angepasst.

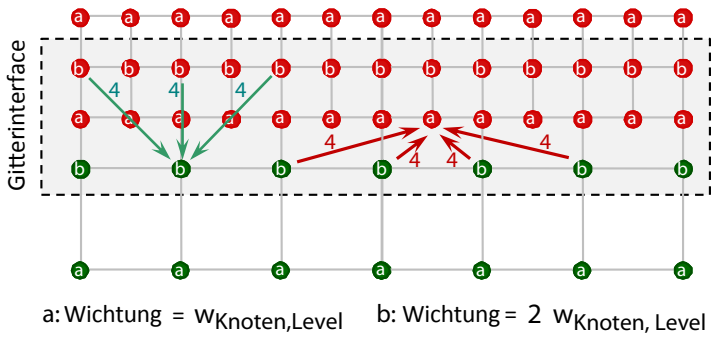


Abbildung 10.5: *Exemplarische Wichtungen für den METIS-Graphen im Gitterübergangsbereich*

Nachdem das Knotengitter in einen METIS-Graphen transformiert wurde, wird die Gebietszerlegung initialisiert. Eine typische METIS-Partionierung sieht z. B. wie folgt aus:

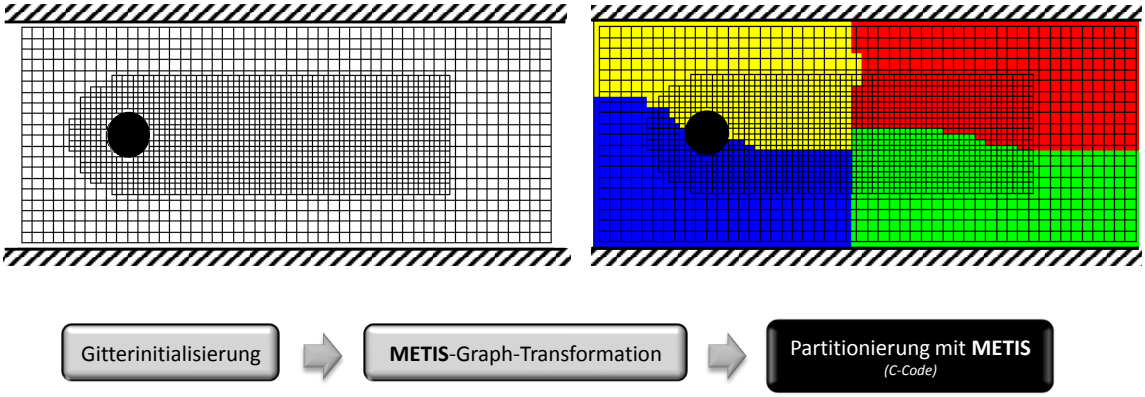
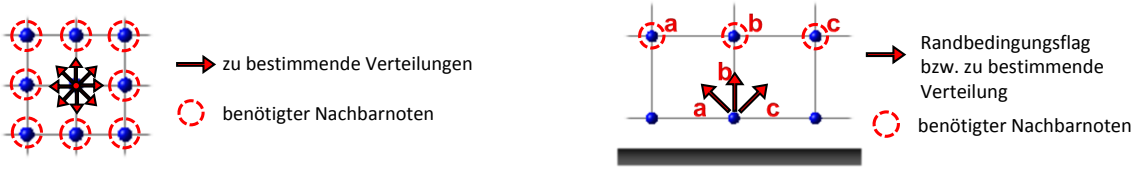


Abbildung 10.6: *Gebietszerlegung mit METIS*

10.4 Haloknoten-Regelwerk

Neben der gleichmäßigen Arbeitsaufteilung muss auch die Kommunikation zwischen den Teilgebieten optimiert werden. Bei Mehrphasenberechungen benötigt man z. B. zur Bestimmung des Phasenfeldgradienten die Phasenfeldinformationen der Nachbarknoten (Abb. 10.7a), die sich jedoch u. U. an Segmenträndern eines anderen Prozesses befinden. Um eine zusätzliche, zeitintensive Interprozesskommunikation zu vermeiden, werden die fehlenden Knoten als sogenannte Haloknoten dem jeweiligen Gebiet zur Verfügung gestellt. Gleiches wird auch bei der Boundary-Fitting-Randbedingung (Abb. 10.7b) durchgeführt.



(a) *Diskretisierungsstern für den Phasenfeldgradienten* ***(b)*** *Boundary-Fitting-Randbedingung*

Abbildung 10.7: *Beispiele für Informationsnachbarknoten*

Halo- bzw. Geisterknoten sind Kopien von Knoten, die einem anderen Prozess zur Verfügung gestellt werden. Der Ursprungsknoten bleibt **einmalig** im Gesamtsystem. Er kann jedoch eine Vielzahl von Haloknoten für verschiedene Prozesse besitzen (*Beispiel:* überlappende Diskretisierungssterne mehrerer angrenzender Teilgebiete).

Die Anzahl notwendiger Geisterknoten hängt von den verwendeten Randbedingungen, Skalierungssternen und dem Strömungsmodell ab. Um dies zu berücksichtigen, wurde ein allgemeingültiges Haloknoten-Regelwerk implementiert. Mit Hilfe benutzerdefinierter Regeln wird entschieden, unter welchen Bedingungen und an welchen Positionen benötigte Nachbarknoten fehlen und somit als Haloknoten hinzugefügt werden müssen. Dadurch wird gewährleistet, dass maximal einmal pro innerstem Zeitschritt zwischen den Prozessen Daten ausgetauscht werden und nötige Änderungen der seriellen Algorithmen auf ein Minimum reduziert werden.

Diese Regeln werden dem Präprozessor der Einfachheit halber in Form von XML-Dateien hinzugefügt. Die Haloknoten, die für eine konsistente Bestimmung des Phasenfeldgradienten notwendig sind, werden beispielsweise mit folgender Regel ermittelt:

Quellcode 10.1: *Phasenfeldgradient-Regel (XML-Snippet)*

```
<LBM_GHOSTNODE_RULE>
   <name> Phasenfeldgradient_Regel </name>
   <level> finest </level>
   <ghostnodes>
      <noncollghostnode>  1  0 </noncollghostnode>
      <noncollghostnode>  1  1 </noncollghostnode>
      <noncollghostnode>  0  1 </noncollghostnode>
      <noncollghostnode> -1  1 </noncollghostnode>
      <noncollghostnode> -1  0 </noncollghostnode>
      <noncollghostnode> -1 -1 </noncollghostnode>
      <noncollghostnode>  0 -1 </noncollghostnode>
      <noncollghostnode>  1 -1 </noncollghostnode>
   </ghostnodes>
</LBM_GHOSTNODE_RULE>
```

Diese enthält, neben dem Regelnamen (`Phasenfeldgradient_Regel`) und dem Level, in dem die Regel zur Anwendung kommen soll (`finest`), die Positionen der benötigten Nachbarn sowie der ggf. zu assoziierende Typ des Haloknotens. Die Position des Nachbarn wird dabei in Relation zu den aktuellen Knotenindizes und Level angegeben (vgl. Abschn. 6.1). Bei `<noncollghostnode> 1 0 </noncollghostnode>` soll dem Segment bei Nichtvorhandensein eines Nachbarn an der Position mit $\Delta\text{Index}_{x_1} = 1$ und $\Delta\text{Index}_{x_2} = 0$ bezogen auf die Knotenindizes des betrachteten Knoten ein Geisterknoten vom Typ **noncollghostnode** hinzugefügt werden.

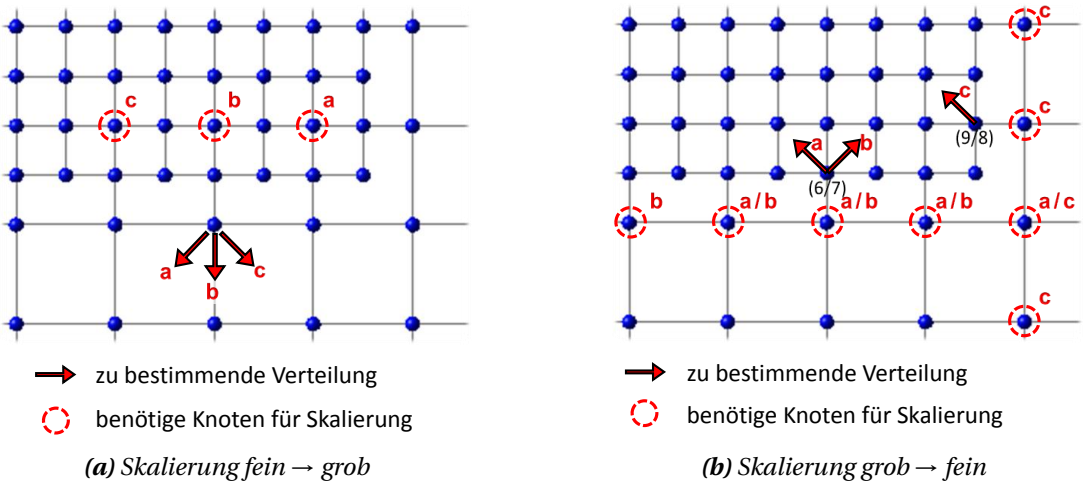
Quellcode 10.2: *Boundary-Fitting-Regel (XML-Snippet)*

```
<LBM_GHOSTNODE_RULE>
   <name> BoundaryFitting_Regel </name>
   <level> all </level>
   <flagname> noslipboundary </flagname>
   <ghostnodes>
      <collghostnode> EAST       1  0 </collghostnode>
      <collghostnode> NORTHEAST  1  1 </collghostnode>
      <collghostnode> NORTH      0  1 </collghostnode>
      <collghostnode> NORTHWEST -1  1 </collghostnode>
      <collghostnode> WEST      -1  0 </collghostnode>
      <collghostnode> SOUTHWEST -1 -1 </collghostnode>
      <collghostnode> SOUTH      0 -1 </collghostnode>
      <collghostnode> SOUTHEAST  1 -1 </collghostnode>
   </ghostnodes>
</LBM_GHOSTNODE_RULE>
```

Bei flagbezogenen Regeln, wie der `BoundaryFitting_Regel` (Quellcode 10.2), erfolgt die Nachbarprüfung nur dann, wenn der betrachtete Knoten in die betreffende Richtung die geforderte Randbedingungsmarkierung (`noslipboundary`) aufweist. Dadurch wird sichergestellt, dass nur die wirklich benötigten Haloknoten dem Gitter hinzugefügt werden.

Infolge der Boundary-Fitting-Implementierung müssen diese Haloknoten, im Gegensatz zu denen der Phasenfeldgradient-Regel, relaxiert werden können. Aus diesem Grund können Haloknoten als

Kollisions- oder Nicht-Kollisions-Haloknoten (collghostnode oder noncollghostnode) markiert werden (Quellcode 10.2).



(a) *Skalierung fein → grob* ***(b)*** *Skalierung grob → fein*

Abbildung 10.8: *Notwendige Knoten für die Gitterskalierung*

Regeln für Haloknoten können beliebig komplex werden. Der in der Skalierungsroutine benötigte Stern für die kubische Rauminterpolation (vgl. Abschn. 3.3) hängt sowohl von der Knotenposition als auch von der Richtung der zu interpolierenden Information ab. Wie in Abb. 10.8b dargestellt, weisen ***a*** und ***c*** die gleiche Richtung auf, verwenden aber eine andere Nachbarschaftsschablone zur Durchführung der Skalierung. Das entscheidende Kriterium ist dabei die Position des Knotens: die Koordinaten für ***a*** sind (x_1=gerade, x_2=ungerade) und für ***c*** (x_1=ungerade, x_2=gerade). Dies führt zu einer komplexen, aber realisierbaren Regel.

Einige Regeln hängen implizit voneinander ab. Für den Fall, dass ein Kollisions-Geisterknoten benötigt wird, müssen auf diesem alle gesetzten Kollisionsregeln rekursiv angewandt werden. Ein Beispiel einer solchen nicht trivialen, rekursiven Anwendung von Regeln, bei der die Phasenfeldgradienten-Regel nur im feinsten Gitter zum Einsatz kommt, ist in Abb. 10.9 dargestellt.

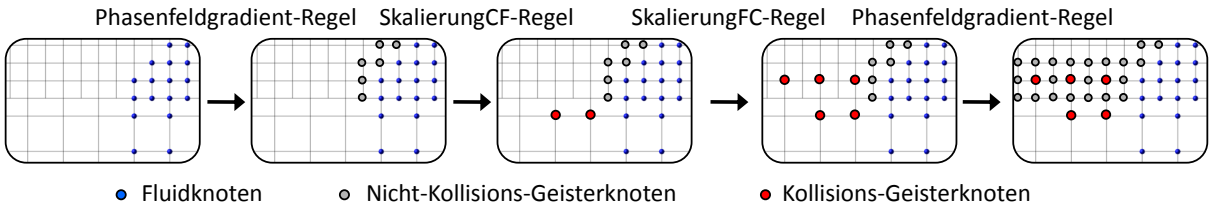
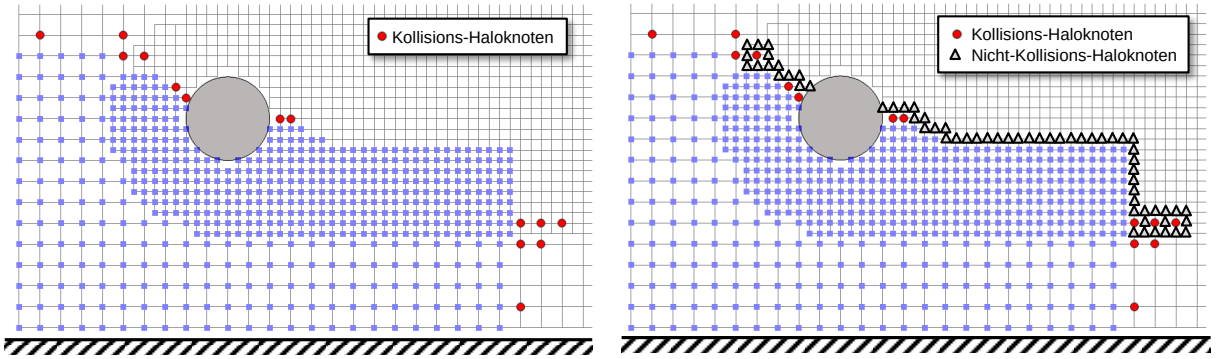


Abbildung 10.9: *Rekursive Regelzuweisung*

In Abb. 10.10 sind als konkrete Beispiele für die Anwendung des Regelwerks zwei Segmente mit den zugehörigen Haloknoten dargestellt. Abb. 10.10a zeigt die Anwendung der Boundary-Fitting-Regel für alle Gitterlevel. In Abb. 10.10b wurde für dasselbe Teilgitter zusätzlich die Phasenfeldgradienten-Regel für den feinsten Level aktiviert.



(a) *Einphasenproblem* ***(b)*** *Mehrphasenproblem*

Abbildung 10.10: *Beispiele für Segmente mit Haloknoten*

10.5 Sende-/Empfangspuffer

Im letzten Schritt des Präprozesses werden die Sende- und Empfangspuffer für den Datenaustausch während der Berechnung optimiert. Aufgrund der Latenzzeit beim Erstellen einer Netzwerkverbindung zwischen zwei Prozessen ist von einer knotenweisen Übertragung abzusehen. Stattdessen werden die zu einem Zeitpunkt auszutauschenden Daten in einem langen, zusammenhängenden Datenvektor (Sendepuffer) gesammelt, in Form einer Nachricht in den Empfangspuffer des Nachbarprozesses übertragen und anschließend auf die Zielknoten verteilt.

> ⚠ Besitzt ein Ziel mehrere Quellen, so ist sicherzustellen, dass die Information von genau *einer* Quelle übertragen wird. Dadurch werden Fehler vermieden, die bei mehrfacher Übertragung entstehen (z. B. doppelte Aufsummierung derselben Phasenfeldinformation bei der Phasenfeldpropagation).

Es wird zwischen zwei Arten des Interprozessdatentransfers unterschieden:

- *geometrisch nicht-lokaler Interprozessdatentransfer* (Abb. 10.11 I-III):
  Die Information wird zu einem geometrischen Nachbarknoten eines anderen Prozesses übertragen (z. B. Verteilungsfunktionen f_{E-SE})
- *geometrisch lokaler Interprozessdatentransfer* (Abb. 10.11 IV):
  Die geometrische Position des Zielknotens entspricht der des Quellknotens (z. B. Übertragung der Verteilungsfunktion f_Z zu einem Haloknoten, die keine eigenständige Kollision durchführen)

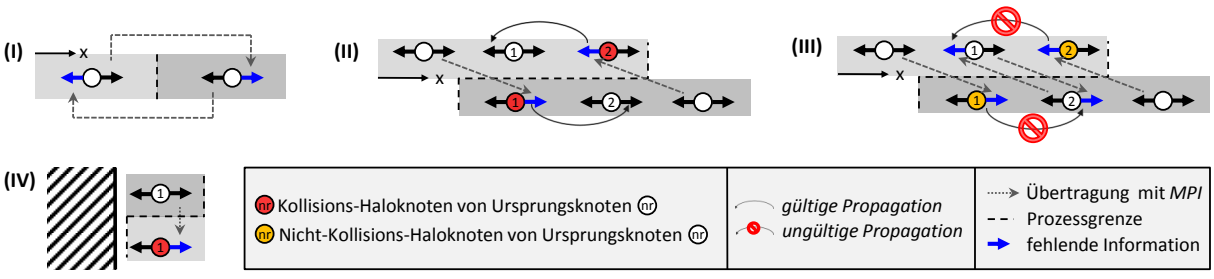


Abbildung 10.11: *Nicht-lokaler Datentransfer (I), (II), (III) und lokaler Datentransfer (IV)*

Während dieses zeitaufwändigen Vorgangs werden die entsprechenden Knoten gekennzeichnet und die Reihenfolge der Sende- und Empfangspuffer festgelegt. Sobald alle notwendigen Informationen vorhanden sind, werden die Teilgebiete zu den jeweiligen Clientprozessen gesendet. Dort werden die speziellen Knotengitter initialisiert und die entsprechenden strömungsdatenenthaltenen Knotendeskriptoren zugewiesen. Für die Initialisierung der Geometrien und Randbedingungen werden dieselben Module wie beim MasterNodeGrid verwendet. Der einzige Unterschied ist, dass den Knoten hier zusätzlich spezielle Randbedingungseigenschaften, wie z. B. Einfluss, Ausfluss oder no-slip-Wand und **q**, zugewiesen werden.

10.6 Präprozess-Speicherbedarf

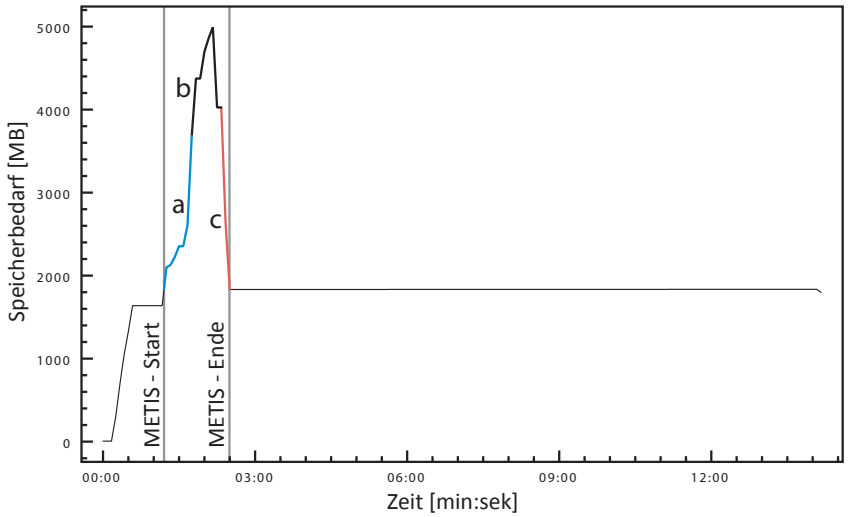


Abbildung 10.12: *Speicherbedarf des Präprozesses*

In Abb. 10.12 ist der charakteristische Speicherbedarf des Präprozesses für eine nicht-uniforme Einphasenströmungssimulation mit ca. zehn Millionen Knoten abgebildet. Die Gebietszerlegung erfolgte für zehn Teilgebiete. Ein Großteil des benötigten Speichers entfällt hierbei auf die Anwendung von METIS. Abschnitt **a** repräsentiert die Konvertierung des Knotengitters in den METIS-kompatiblen Graphen, der zur Partitionierung (Abschnitt **b**) benötigt wird und in **c** aus dem Speicher entfernt wird. Dieser kurzzeitig sehr hohe Speicherbedarf schränkt die Anwendung auf große, komplexe Gitter stark ein. Ein möglicher Ausweg ist die Gruppierung der Knoten in einzelne, diskrete Blöcke, für die anschließend die eigentliche Segmentierung durchgeführt wird. Dieser Ansatz würde weniger Speicher und weniger Zeit benötigen. Alternativ kann man an Stelle der METIS-Bibliothek auch die parMETIS- oder eine andere Partitionierungsbibliothek verwenden. Hierfür stellt das MasterNodeGrid eine entsprechende Schnittstelle zur Verfügung.

11 Validierung III (Ein-/Mehrphase, verteilt)

Die Validierungsrechnungen des verteilten Knotengitteransatzes in VIRTUALFLUIDS wurden auf dem iRMB-PC-Cluster mit 47 Dual-Prozessor-Knoten (2x64-Bit-*AMD-Opteron*™ mit 1,4 GHz), die untereinander über ein *Myrinet 2000*-Netzwerk kommunizieren, durchgeführt. Sowohl die parallele Effizienz als auch die Skalierbarkeit wurden für verschiedene Testkonfigurationen für Ein- und Mehrphasenprobleme untersucht. Soweit möglich, wurden die Rechenclients separaten Clusterknoten zugewiesen, um u. a. den maximalen Arbeitsspeicher ausnutzen zu können. Es wurden zwei verschiedene 2-D-Rechenkerne des VIRTUALFLUIDS-Paketes untersucht. Der *MultiPhase*-Kern kann sowohl für Ein- als auch für Mehrphasensimulationen verwendet werden. Die spezifischen Knotendeskriptoren werden in dem in Abschn. 6.2 beschriebenen ChessMemPool2D gespeichert, um durch die erhöhte Datenlokalität die Zugriffe auf den Hauptspeicher zu minimieren. Im Gegensatz hierzu speichert der hinsichtlich performanter Einphasenberechnungen optimierte *LbVector*-Kern die Verteilungen nicht in den Deskriptoren, sondern mittels indirekter Adressierung in eindimensionalen Feldern (vgl. Abschn. 6.2).

Als Betriebssystem stand die 64-Bit-Version von Debian 3.1 [26] zur Verfügung und die Übersetzung des Codes wurde mit der GNU Compiler Collection Version 3.4.4 (Optimierungsoption: 03) [44] durchgeführt. Die Kommunikation über das Myrinetnetzwerk erfolgte mit Hilfe der speziell dafür angepassten MPICH1-Bibliothek [107].

Einen Überblick über die Performance der einzelnen Pakete gibt Tab. 11.1 in Knotenaktualisierungen pro Sekunde (*nups, engl.: nodal updates per second).*

Berechnungskern	uniform	nicht-uniform
LbVector	3,0E6	2,7E6
$\text{MultiPhase}_{Einphasenmodus}$	0,9E6	0,85E6
$\text{MultiPhase}_{Mehrphasenmodus}$	0,6E6	0,55E6*

* Verhältnis von Mehrphasen- zu Einphasenknoten = $\frac{1}{10}$

Tabelle 11.1: *nups auf einer 64-Bit-AMD-Opteron™ CPU (1.4 GHz)*

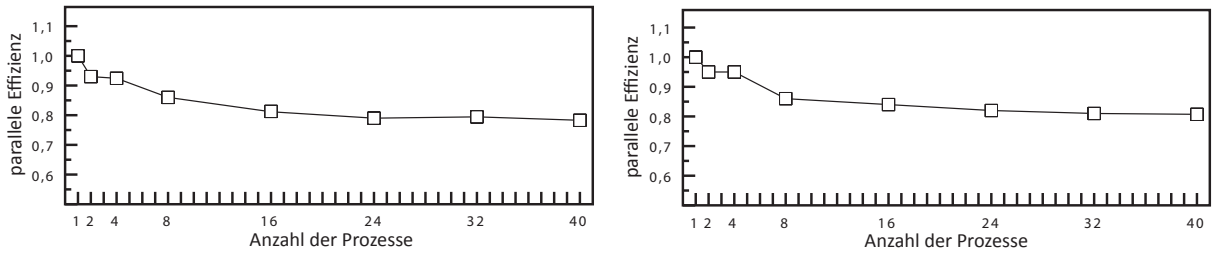
11.1 Parallele Effizienz

Die Effizienz eines parallelen Codes steht u. a. im engen Zusammenhang mit der Anzahl an zusätzlichen Operationen die gegenüber der seriellen Version auszuführen sind und mit der Qualität der Lastverteilung (Gl. 11.1).

$$\textit{parallele Effizienz} = \frac{\textit{Rechenzeit mit } \boldsymbol{einem} \textit{ Rechenprozess}}{\textit{Rechenzeit mit } \boldsymbol{n} \textit{ Rechenprozessen} \cdot \boldsymbol{n}} \tag{11.1}$$

Ein entscheidender Faktor ist das Verhältnis der Segmentkantenlänge zur Gitterknotenanzahl (Freiheitsgrade) der Teilgebiete. Die Kantenlänge repräsentiert hierbei den Aufwand der Interprozesskommunikation. Erhöht man die Zahl der Gitterknoten pro Segment bei gleich bleibender Anzahl

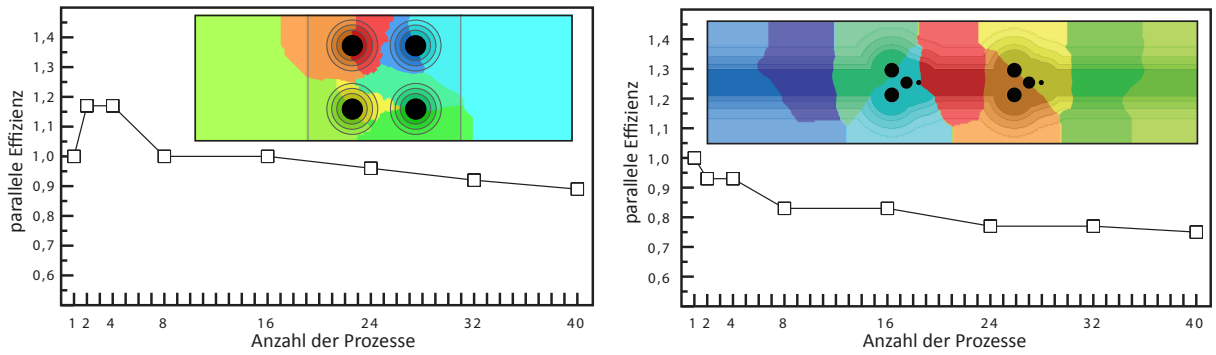
von Prozessen, so resultiert dies im Regelfall in einer höheren parallelen Effizienz, wenn genügend Hauptspeicher zur Verfügung steht. Mit wachsender Zahl an Prozessen bei gleich bleibender Anzahl an Gitterknoten hingegen steigt das Verhältnis der lokalen Kantenlänge zur Anzahl an Gitterknoten, und resultiert in einer verminderten parallelen Effizienz.



(a) *Einphasenmodus mit* $5 \cdot 10^6$ *Knoten* ***(b)*** *Mehrphasenmodus mit* $5 \cdot 10^6$ *Knoten*

Abbildung 11.1: *Parallele Effizienz des MultiPhase-Kerns (uniform)*

Im uniformen Fall wurde eine Spaltströmung ohne Hindernisse mit no-slip-Wänden (Boundary-Fitting) sowie Ein- und Ausflussrandbedingungen gerechnet. Bei den Simulationen wurde pro Clusterknoten nur eine CPU verwendet. Zudem war der CPU-eigene Arbeitsspeicher für die jeweilige Berechnung ausreichend dimensioniert. Aufgrund der höheren seriellen Leistung des *MultiPhase*-Kerns im Einphasenmodus benötigt dieser für die Berechnung der einzelnen Teilgebiete im Vergleich zum Mehrphasenmodus weniger Rechenzeit (vgl. Tab. 11.1). Aus diesem Grund ist die parallele Effizienz für den schnelleren Kern etwas geringer, da der zeitliche Mehraufwand für den Interprozessdatenaustausch bei verteilten Berechnungen einen höheren Einfluss hat (Abb. 11.1a und Abb. 11.1b). Dennoch ist der Unterschied nicht so signifikant wie bei einer 30 % höheren seriellen Leistung erwartet. Gründe hierfür sind die höhere Anzahl zu berücksichtigender Haloknoten pro Teilgebiet (vgl. Abschn. 10.4), die erhöhte zu kommunizierende Datenmenge bei Mehrphasenproblemen und das vorhandene, sehr effiziente Netzwerk.



(a) *Einphasenmodus mit* $5 \cdot 10^6$ *Gitterknoten* ***(b)*** *Mehrphasenmodus mit* $2 \cdot 10^6$ *Gitterknoten*

Abbildung 11.2: *Parallele Effizienz des MultiPhase-Kerns (nicht-uniform)*

Bei der Messung der parallelen Effizienz für nicht-uniforme Gitter, in denen Teilbereiche vor allem im Bereich von Festkörpern verfeinert wurden, musste infolge einer Speicherreduzierung auf den Clusterknoten zur Bestimmung des Referenzwerts der Gesamtspeicher eines Clusterknoten verwendet werden. Der dadurch in Abb. 11.2a zu beobachtende superlineare Speedup ist ein Effekt, der auf die verwendete cache-coherent Non-Uniform Memory Architecture (ccNUMA) des PC-Clusters zurück-

zuführen ist. Hierbei besitzt jeder Prozessor seinen eigenen Speicher kann aber über den sehr schnellen Hypertransport auf den der zweiten CPU zugreifen (Abb. 11.3). Zugriffe auf den Speicher einer anderen Lokalitätsdomäne führen trotz schneller Netzwerke zu signifikanten Performanceeinbußen [31, 142]. Dieser indirekte Zugriff ist im Vergleich zum direkten ineffizienter, weshalb die serielle Leistung in diesem Testfall gegenüber Simulationen bei denen eine CPU ausschließlich auf den eigenen Speicher zugreift geringer ausfällt. Ab der Verwendung von zwei CPUs, die sich jeweils auf einem anderen Clusterknoten befanden, war der jeweils zur Verfügung stehende CPU-eigene Arbeitsspeicher ausreichend bemessen, sodass keine zeitintensiven Zugriffe über Hypertransport mehr notwendig waren und die erzielte Effizienz über dem seriellen Referenzwert lag. Bei den Testfällen mit $2 \cdot 10^6$ Berechnungsknoten kam der Hypertransport nicht zum Einsatz, weshalb der superlineare Speedup dort nicht auftrat (Abb. 11.2b).

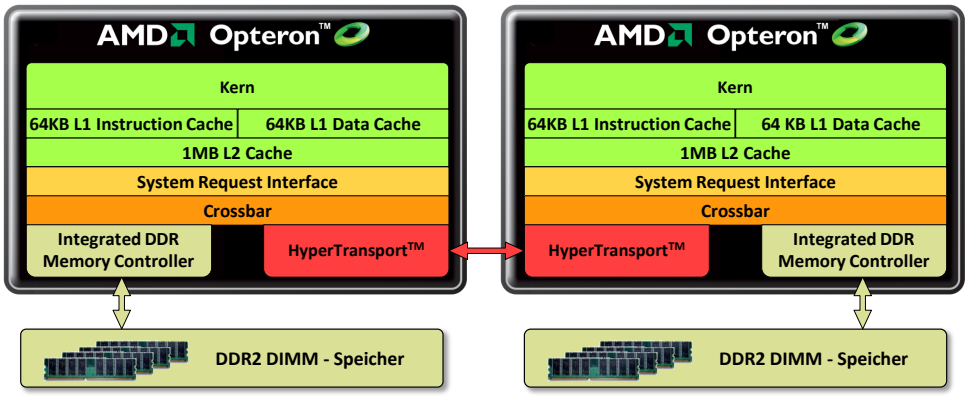


Abbildung 11.3: *ccNUMA Architektur bei zwei 64-Bit-Einkern-AMD-Opteron™-CPUs*

Zur Validierung des *LbVector*-Lösers wurde eine Tandemzylinderströmung berechnet. Für den nicht-uniformen Testfall wurde das Gebiet um die Zylinder als auch in deren Nachlauf verfeinert (vgl. Abb. 11.4). Alle Testgebiete wurden hier so konfiguriert, dass ein ccNUMA Effekt vermieden wurde.

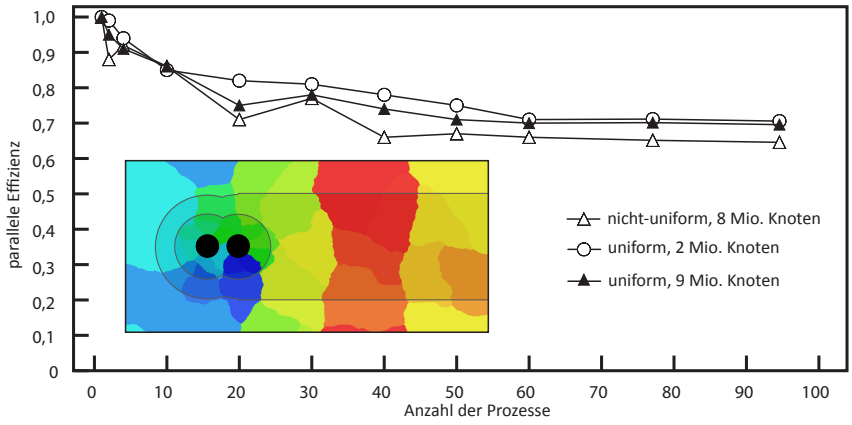


Abbildung 11.4: *Parallele Effizienz des LbVector-Kerns für uniforme und nicht-uniforme Gitter*

Simulationen mit nicht-uniformem Gitter erzielten beim *LbVector*-Kern eine geringere parallele Effizienz als die mit uniformen Gittern. Eine Ursache hierfür ist die, aufgrund der starren Datenstruktur des Berechnungskerns (Abschn. 6.2), nicht optimierte Implementierung der Füll- und Verteilungs-

algorithmen der zu übertragenden Informationen. Gemessen an der im Vergleich hohen, seriellen Leistungsfähigkeit sind die Ergebnisse bereits sehr zufriedenstellend.

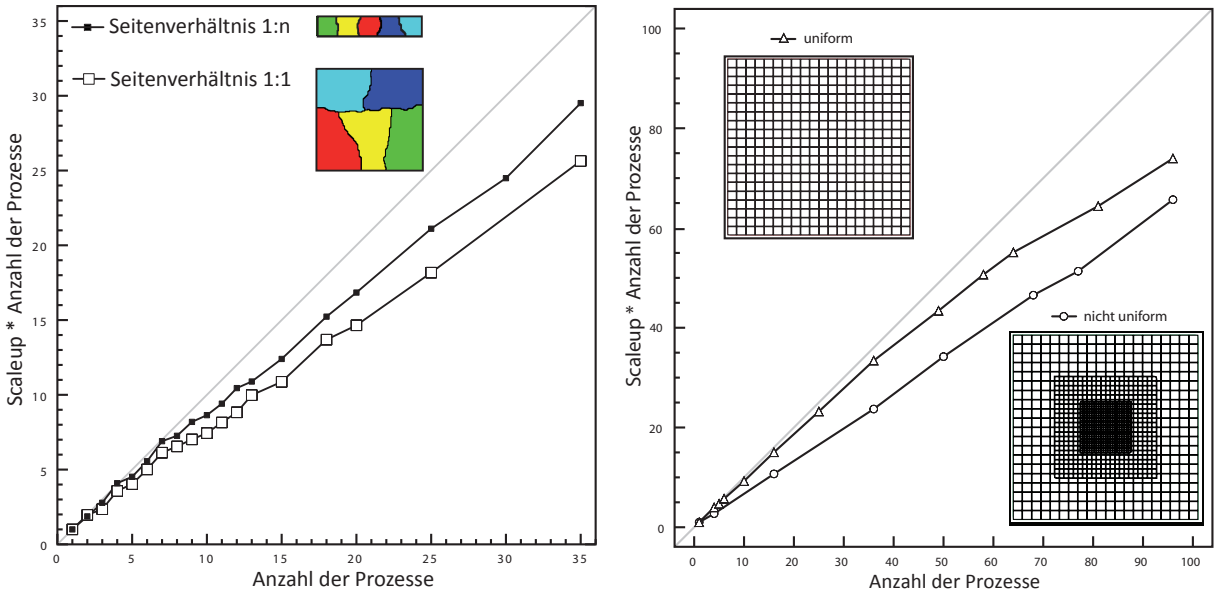
Die Güte der erzielten Lastverteilung für verschiedene Prozessanzahlen wirkt sich unterschiedlich auf die Effizienz aus. Dies ist insbesondere bei nicht-uniformen Gittern, bei denen heuristische Knoten- und Kantenwichtungen (Randbedingungs-, Gitterübergangsknoten, etc.) verwendet werden, um die Qualität der mit METIS durchgeführten Partitionierung zu steuern, sichtbar (Abb. 11.2, 11.4). Zudem führten schwer zu beeinflussende Effekte durch die vorhandene Netzwerktopologie, die u. a. höhere Latenzzeiten zur Folge hatten, zu einer nachhaltigen Minderung der parallelen Effizienz.

11.2 Skalierbarkeit

Um die Skalierbarkeit eines Systems zu ermitteln, erhöht man proportional die Anzahl an Gitterknoten zu der Anzahl an Rechenprozessen (Gl. 11.2).

$$Scaleup = \frac{Rechenzeit\ mit\ \mathbf{einem}\ Rechenprozess\ mit\ \mathbf{m}\ Freiheitsgraden}{Rechenzeit\ mit\ \mathbf{n}\ Rechenprozessen\ mit\ je\ \mathbf{m}\ Freiheitsgraden} \tag{11.2}$$

Der Scaleup wurde für rechteckige, in beide Achsrichtungen periodische Gitter bestimmt. Die Ausgangsgitter für Berechnungen mit einer CPU enthielten dabei 50.000 Rechenknoten. Beim *MultiPhase*-Kern und Simulationen auf uniformen Gittern wurde die Rechenzeit von Gebieten mit unterschiedlichen Seitenverhältnissen und bei gleicher Gitterknotenanzahl verglichen, wodurch mögliche Effekte der Gebietszerlegung gezeigt werden konnten.



(a) *Uniforme Gitter mit unterschiedlichen Profilen (MultiPhase-Kern)*

(b) *Uniforme und nicht-uniforme Gitter (LbVector-Kern)*

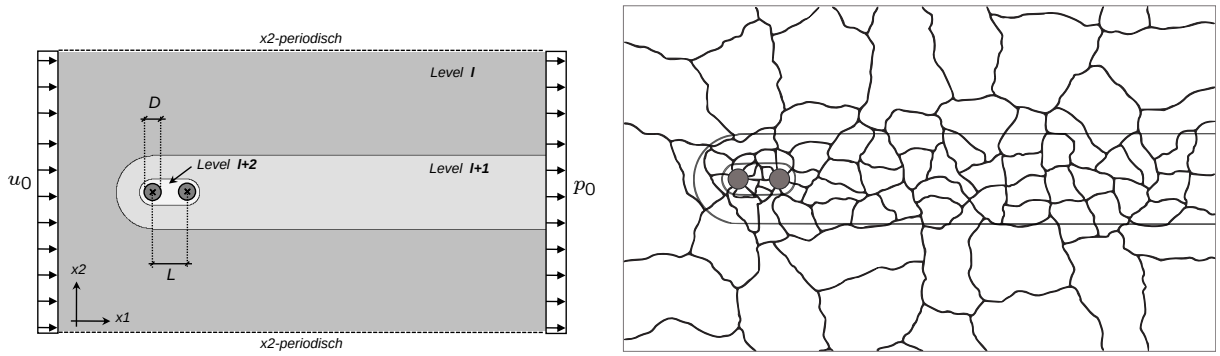
Abbildung 11.5: *Scaleup-Ergebnisse*

Wie in Abb. 11.5a zu erkennen, sinkt die Effizienz erwartungsgemäß mit steigender Kommunikationskomplexität. Aufgrund der METIS-Segmentierung weisen die nicht-quadratischen Gitter infolge

der kürzeren Interprozesskommunikationskanten und geringeren Nachbarprozessanzahl eine höhere Effizienz auf.

Da Ingenieursprobleme typischerweise mit verschieden feiner Gitterauflösung gerechnet werden, die bei entsprechend erhöhter CPU-Zahl idealerweise dieselbe Zeit beanspruchen sollten, wurde bei den nicht-uniformen Testfällen anders als sonst üblich die Gitterknotenanzahl der Basisgitterkonfiguration levelweise skaliert anstatt das Gitter als solches entsprechend der CPU-Zahl zu vervielfältigten. Die dabei erzielte Skalierung lag beim *LbVector*-Kern unterhalb der uniformer Gitter (Abb. 11.5b). Eine Ursache hierfür ist die Wahl der Knoten- und Kantenwichtungen, die hier zu einer unausgewogenen Lastverteilung führten. Die Verwendung anderer Wichtungen oder eines anderen Gebietszerlegers könnte hier zu einer besseren Skalierung führen.

11.3 Tandemzylinderströmung für Reynoldszahl 1.000



(a) *Setup* ***(b)*** *METIS-Gebietszerlegung für 78 Prozesse*

Abbildung 11.6: *Konfiguration für den Tandemzylindertestfall mit* $\frac{L}{D} = 2,15$

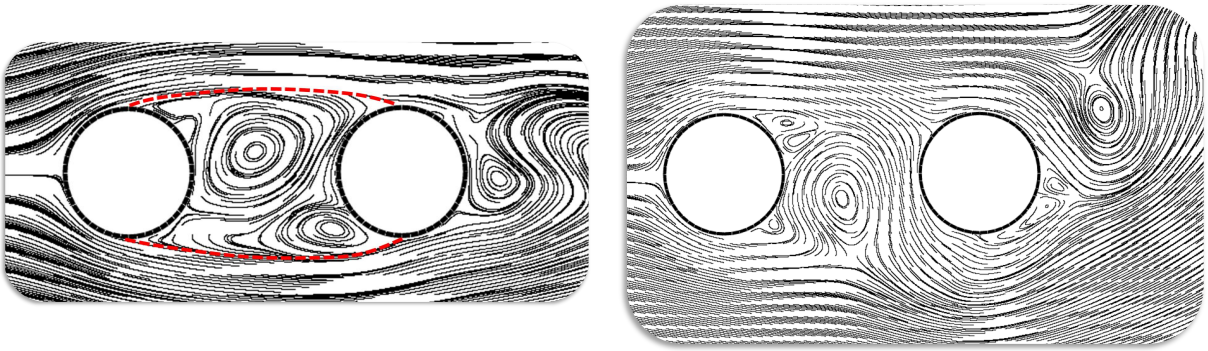
Der Tandemzylindertestfall ist ein sehr anspruchsvolles CFD-Benchmarkproblem. Trotz der geometrisch einfachen Konfiguration (Abb. 11.6a) kann hier eine Vielzahl von Strömungsphänomenen beobachtet werden, die z. B. bei Ölplattformen, Kühltürmen oder Fahrwerken landender Flugzeugen [84] auftreten.

In der Literatur findet man oft Simulationen für kleine Reynoldszahlen ($Re = \frac{|\mathbf{u}| \cdot D}{\nu}$) im Bereich von 5 bis 200 [30, 139, 145]. Numerische Berechnungen für Re > 20.000 basieren meist auf Turbulenzmodellen.

Um den Vorteil der Parallelisierung zu nutzen, wurde eine Simulation mit einer Reynoldszahl von 1.000 und einem L/D-Verhältnis von 2,15 durchgeführt. Das Gebiet wurde im Bereich der Zylinder und deren Auslauf, in dem sich eine Wirbelstraße entwickelt, verfeinert (Abb. 11.6a).

Die Zylinderauflösung beträgt 570 Knoten, sodass eine direkte numerische Simulation (DNS) ohne Verwendung eines Turbulenzmodells durchgeführt werden kann. Zur Simulationslaufzeit treten zwei aus der Literatur [106] und anderen Simulationen bekannten Regime auf. Zu Beginn wird das sogenannte *Reattachment Regime (RTR)* beobachtet, bei dem sich die Stromlinien des stromaufwärts liegenden Zylinders ablösen und am zweiten Zylinder wieder anlegen (Abb. 11.7a). Später lösen sich diese Stromlinien vom zweiten Zylinder, und das *Two Vortex Street Regime (TVS)* tritt ein. Dies wurde nach der sich einstellenden Wirbelstraße hinter den Zylindern, die wechselweise zwischen den

beiden Zylindern erzeugt wird, benannt (Abb. 11.7b). Die Berechnungen erfolgten mit dem *LbVector*-Kern in Kombination mit dem MRT-Modell (vgl. Abschn. 3.1).



(a) *Reattachment Regime* ***(b)*** *Two vortex street Regime*

Abbildung 11.7: *Regimes einer Tandemzylinderströmung*

Vergleicht man die ermittelten Kraftbeiwerte mit Referenzlösungen auf Basis der Navier-Stokes-Gleichungen [106], so stimmen diese quantitativ überein (Tab. 11.2).

	c_D stromaufwärts		c_D stromabwärts		St	
Regime	LB	NS	LB	NS	LB	NS
RTR	0,93	0,92	-0,32	-0,34	0,162	0,16
TVS	1,38	1,35	0,79	0,78	0,169	0,17

Tabelle 11.2: *Kraftbeiwerte und Strouhal-Zahl der Zylinder*

Der Vorteil des nicht-uniformen Gitters, das für die verteilte Simulation in 78 Teilgebiete unterteilt wurde (Abb. 11.6b), lässt sich gut an der Anzahl der Freiheitsgrade erkennen (Tab. 11.3). Uniform wären es ungefähr $1,5 \cdot 10^8$ Knoten, was nicht nur für den parallelen Löser, sondern auch für den Postprozess eine Herausforderung darstellen würde.

Level	Knoten pro Level	Knoteniterationen pro groben Zeitschritt
11	5.415.466	5.415.466
12	12.247.251	24.494.502
13	4.572.398	18.289.592
11-13	22.235.115	70.164.675

Tabelle 11.3: *Aktive Berechnungsknoten je Gitterlevel*

Teil III

VIRTUALFLUIDS auf hybriden Blockgittern

Überblick

Dieser Teil der Arbeit beschäftigt sich mit der Entwicklung eines komponentenbasierten, verteilt arbeitenden Frameworks für LB-Simulationen. Hierzu wird zunächst eine geeignete Interprozessbibliothek ermittelt, die eine einfache, flexible und vor allem objektorientierte Kommunikation der zu entwickelnden VIRTUALFLUIDS-Komponenten ermöglicht. Anschließend werden die Grundlagen der hybriden Blockgitterstruktur als konsequente Weiterentwicklung des Knotengitteransatzes erläutert sowie auf die notwendige Umstrukturierung des vorhanden Codes eingegangen. Dabei wurde in Hinblick auf einen einfachen Datenaustausch Wert auf eine generalisierte Interblockkommunikation gelegt, die letztlich mit dem entwickelten Connector-Transmitter-Konzept ermöglicht wurde. Im Weiteren wird detailliert auf die Interaktoren eingegangen, die als Schnittstelle zwischen Blockgitter und Festkörpern agieren und aufgrund ihrer umfangreichen Funktionalität ein wesentlicher Bestandteil von VIRTUALFLUIDS sind. Von dieser Basis aus folgt die Beschreibung des implementierten plattformunabhängigen Serviceframeworks. Nach Vorstellung der vorhandenen Dienste wird deren Funktionsweise anhand einer typischen verteilten Simulation sowie die Umsetzung einiger Optimierungen ausführlich erklärt. Anschließend wird auf die Rechenleistung des Prototypen eingegangen und diskutiert. Zum Abschluss des Teils werden einige mit VIRTUALFLUIDS berechnete Anwendungsbeispiele präsentiert.

12 Motivation

Der knotenbasierte Ansatz erfüllt die in Kap. 6 geforderten Anforderungen. Die flexible, generalisierte Datenstruktur lässt sich jederzeit für andere Löser erweitern, und die implementierten Algorithmen ermöglichen eine effektive Gittergenerierung sowie eine dynamische Gitteranpassung zur Berechnungslaufzeit. Bei verteilten Simulationen können mit Hilfe des entwickelten Regelwerkes beliebige Diskretisierungssterne berücksichtigt werden.

Gegenüber matrixbasierten Lösern ist der Speicherbedarf der Knotengitter jedoch verhältnismäßig hoch und die Rechenleistung ohne Verwendung der Vektorstruktur vor allem in 3-D nicht ideal. Zudem erwies sich, trotz guter paralleler Leistung, die Umsetzung einer Kombination des adaptiven und verteilten Knotenansatzes als zu komplex und ineffizient. Gründe hierfür waren unter anderem der zeit- und speicherintensive Präprozess, die zu berücksichtigenden Knotenkonstellationen bei der Gebietszerlegung und die mangelnde dynamische Interprozesskommunikation seitens MPI.

Aus diesem Grund musste zunächst eine geeignete Interprozesskommunikationsbibliothek gefunden werden, die beliebige Anfragen zwischen den einzelnen Komponenten sowie die einfache Übertragung von Objekten zulässt. Danach wurden Überlegungen angestellt, wie das vorhandene Konzept modifiziert werden muss, um die vorhandenen Defizite weitestgehend zu beseitigen ohne gleichzeitig die Flexibilität des Knotenansatzes zu reduzieren. Der Fokus wurde dabei auf ein verteilt arbeitendes, komponentenbasiertes Framework mit dynamisch adaptiver Gittersteuerung gelegt.

13 Dynamische Kommunikation in verteilten Systemen

Das zu erstellende Framework soll auf verteilten Systemen wie PC-Clustern, Firmennetzwerken oder Großrechnern zum Einsatz kommen. Für die notwendige Interprozesskommunikation (IPC) innerhalb solcher Systeme existiert eine Vielzahl von Lösungen. In diesem Kapitel werden deshalb das bereits in Kap. 10 verwendete MPI sowie verschiedene Middleware-Konzepte hinsichtlich ihrer Anwendbarkeit in einer verteilten, komponentenbasierten C++-Anwendung überprüft. Dabei wird insbesondere auf die Möglichkeit einer, für den adaptiven Löser notwendigen, dynamischen Interaktion auf Basis des Client-Server-Modells sowie der Übertragung komplexer Objekte eingegangen.

13.1 MPI

Bei der Parallelisierung des Knotengitters hat sich gezeigt, dass mit MPI [134] eine effektive Kommunikationslösung zum Austausch numerischer Daten zwischen mehreren Prozessen während der Berechnung existiert. Zum Datenaustausch werden dabei Nachrichten verwendet, die in einer zuvor festgelegten Reihenfolge zu einem anderen Prozess gesendet werden. Diese Daten bestehen vorwiegend aus Feldern mit primitiven Datentypen.

Ist man jedoch an einem dynamischen Austausch von komplexen Objekten interessiert, wie sie in einer objektorientierten Sprache vorkommen, stößt man sehr schnell an die Grenzen des MPI-Standards. Um selbst definierte Objekte versenden zu können, müssen diese zuvor mit Hilfe der MPI-Funktionen *MPI_Pack* und *MPI_Unpack* in die entsprechenden Puffer geschrieben bzw. aus diesen rekonstruiert werden. Insbesondere bei Objekten mit dynamischen Attributen führt dies schnell zu einem unübersichtlichen und schwer zu wartenden Quellcode. Hier kann die Verwendung von Boost::MPI [8] oder TPO++ [64] Abhilfe schaffen. Neben einem C++-Interface ermöglichen diese objektorientierten MPI-Bibliotheken unter entsprechender Klassenerweiterungen das Versenden von beliebigen Objekten mit MPI.

Die meisten MPI-Implementierungen basieren auf prozeduralen Sprachen, wie Fortran und C. Erst mit dem Erscheinen des MPI-2-Standards [105] erfolgte die Spezifikation zusätzlicher Sprachschnittstellen, wie z. B. für C++-Implementierungen, die für spezielle Netzwerkarchitekturen eines Großrechners geschrieben wurden, unterstützen diese jedoch oftmals nur unzureichend.

Ein weiterer Nachteil des MPI-Standards hinsichtlich einer dynamischen Interaktion ist, dass er keine beliebige, dem Client-Server-Modell ähnliche, Kommunikation vorsieht. Eine solche Funktionalität muss eigenständig implementiert werden. Diesbezüglich sind in [101] einige Ansätze beschrieben. In diesem Zusammenhang muss die erhöhte Komplexität von MPI bei Multithreadanwendungen erwähnt werden [59, 62].

13.2 Middleware-Lösungen

Was genau ist Middleware? Die einfachste und prägnanteste Definition lautet:

> *„Common definition are that middleware is the glue between software components or between software and the network or it is the slash in Client/Server.“*
>
> (Quelle: http://www.middleware.org)

In dieser Arbeit repräsentiert *Middleware* Vermittlungssoftware, die den Datenaustausch zwischen verschiedenen Applikationen gewährleistet und verteilte Anwendungen integriert. Sie soll dem Programmierer eine transparente Kommunikation zwischen verschiedenen Prozessen und Rechnern ermöglichen, ohne dass dieser über Kenntnisse von jeweiligen systemspezifischen Low-Level Socket-API oder Interprozesskommunikationsmethoden, wie Sockets oder Memory Mapped Files, verfügen muss. Stattdessen soll die gewohnte Sicht auf Objekte, die über Methodenaufrufe miteinander kommunizieren, erhalten bleiben. Systemübergreifende Anfragen werden über das Netzwerk weitergereicht, wobei gewöhnlich Standardnetzwerkprotokolle wie TCP/IP zum Einsatz kommen. Das Marshalling, also das Umwandeln der Daten in ein einheitliches, über das Netzwerk versendbares Format sowie deren Rücktransformation am Ziel, ist ein wesentlicher Bestandteil einer Middleware-Software.

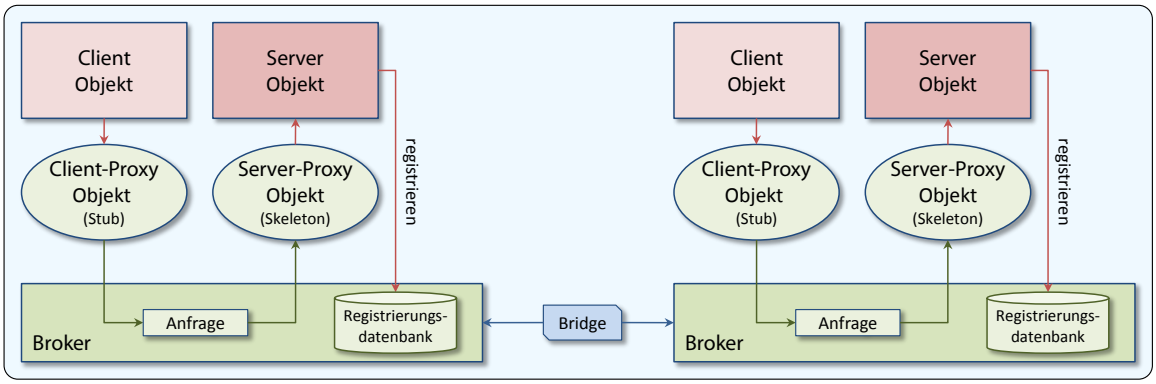


Abbildung 13.1: *Broker-Pattern*

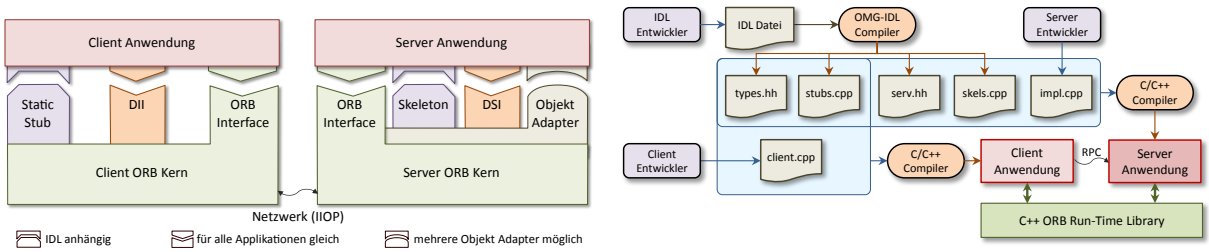
Im Allgemeinen verwendet Middleware zur Kommunikation das Client-Server-Modell (die serverseitige Bearbeitung von Clientanfragen), wobei die Transparenz der Kommunikation durch das Broker-Pattern realisiert wird (Abb. 13.1). Die Brokerkomponente ist hierbei der zentrale Punkt, über den die gesamte Client-Server-Kommunikation abläuft. Dadurch wird die gewünschte Transparenz gewährleistet. Sogenannte Client-Proxys (Stubs) wandeln die Serveranfragen eines Clients an einen Server in das brokerspezifische Format um und leiten sie entsprechend weiter. Die Proxyschnittstellen werden in der Regel über eine Interface Definition Language (IDL) definiert. Der Broker ermittelt den Zielserver und übergibt die Anfrage an den jeweiligen Server-Proxy (Skeleton), der anschließend die ursprüngliche Anfrage an das Zielobjekt weiterleitet. Verschiedene Broker können untereinander optional über Bridges kommunizieren [13, 135, 152].

Im Folgenden werden aktuelle plattformunabhängige und C++-unterstützende Middleware-Lösungen vorgestellt.

13.2.1 CORBA

Bei der Common Object Request Broker Architecture (CORBA) handelt es sich um eine Spezifikation der Object Management Group (OMG) [150], für die es eine Vielzahl von Implementierungen gibt, z. B. TAO [124] und OmniORB [60]. Durch Einhaltung dieses Standards können Prozesse zwischen verschiedenen Programmiersprachen (C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python und IDLscript) und Betriebssystemen (z. B. Windows, Linux) miteinander kommunizieren. In Abb. 13.2a ist die Umsetzung des Broker Patterns in CORBA dargestellt. Seit CORBA 2.0 erfolgt die Kommunikation zwischen den Objekt Request Brokern (ORB) über das TCP/IP basierte Internet Inter ORB Protocol (IIOP), das erstmals die Zusammenarbeit verschiedener CORBA-Implementierungen ermöglichte.

Die Funktionalität der verteilten Objekte wird mit Hilfe der OMG-IDL beschrieben, die große Ähnlichkeit zu C++ aufweist. So stehen dem Anwender u. a. einfache Datentypen (short, string, etc.), zusammengesetzte Datentypen (struct), Namensräume (module), Klassen (interfaces), Vererbung und Ausnahmebehandlung (exceptions) zur Verfügung. Der IDL-Compiler generiert aus diesen Informationen die für die jeweilige Programmiersprache benötigten Stubs und Skeletons, die auf die jeweilige ORB-Implementierung abgestimmt sind [76] (Abb. 13.2b).



(a) Broker-Pattern bei CORBA *(b) Entwicklungprozess*

Abbildung 13.2: *CORBA-Design*

Bevor der Server dem ORB-Objekte einer Klasse zur Verfügung stellen kann, muss deren konkrete Implementierung vorliegen. Hierzu wird sie von der generierten Skeletonklasse abgeleitet und die darin enthaltene, zuvor in der IDL definierten Funktionalität implementiert. Die Skeletonklasse ist ein sogenannter Portable Object Adapter (POA), der die Objekte einer Klasse (Servant) mit den ORB verbindet und dafür sorgt, dass Anfragen an ein bestimmtes CORBA-Objekt an den richtigen Servant weitergeleitet werden.

Die Implementierung des Clients benötigt lediglich die erzeugten Stubinformationen (Abb. 13.2b). Um auf ein entferntes Objekt zugreifen zu können, benötigt der Client zur Laufzeit dessen Interoperable Object Reference (IOR), die sämtliche Informationen enthält, um einen Kommunikationskanal zu einem Objekt aufbauen und entsprechende Aufrufe durchführen zu können.

Ein weiterer Vorteil von CORBA ist, dass bereits eine Vielzahl von Diensten (Object Services) zur Verfügung gestellt wird, die für viele Aufgabenstellungen und Probleme geeignet sind. Beispiele hierfür ist der Naming Service, der von Clients verwendet wird, um verteilte, von Servern angebotene Objekte zu lokalisieren, und der Event Service, der für die asynchrone Kommunikation und Ereignisbehandlung genutzt wird.

Neben den erwähnten statischen Mechanismen bietet CORBA zwei standardisierte Schnittstellen zur Unterstützung dynamischen Verhaltens an: das Dynamic Invocation Interface (DII) und das Dynamic Skeleton Interface (DSI). Beide werden direkt vom ORB unterstützt und sind unabhängig von

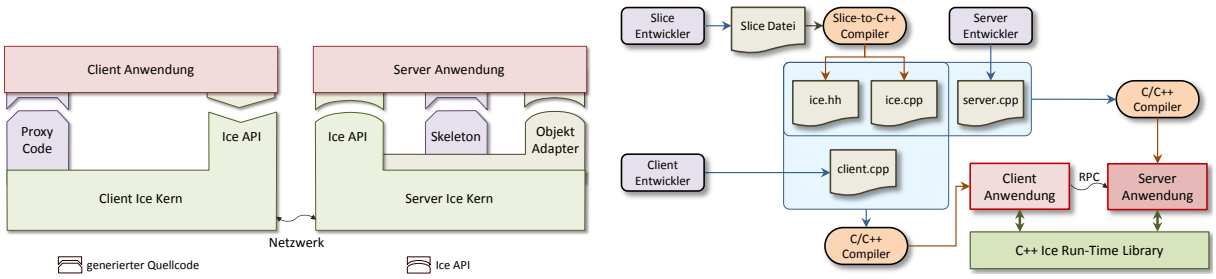
den IDL-Interfaces der aufgerufenen Objekte. Für eine detaillierte Beschreibung und weiterführende Informationen wird auf die Literatur verwiesen [76].

Seit der Entstehung des CORBA-Standards im Jahre 1989 stieg die Komplexität stetig an, insbesondere die der APIs. Dadurch ist selbst die Implementierung eines trivialen Programms sehr aufwändig. Auch das Mapping auf C++ ist relativ schwierig zu handhaben und bereitet Probleme hinsichtlich Threadsicherheit, Speichermanagement und Ausnahmesicherheit [73, 74]. Unterschiedliche (Allokierungs-)Regeln für in, out bzw. inout IDL-Funktionsparameter und unterschiedliche Typen sind oft die Quelle von schwer auffindbaren Fehlern. Ähnlich zu MPI unterstützen CORBA-Implementierungen oftmals nicht sämtliche Spezifikationen des Standards und integrieren meist nur das Mapping für C++ und Java.

13.2.2 Ice

Die Firma ZeroC hat sich zum Ziel gesetzt, die Unzulänglichkeiten anderer Middlewarekonzepte zu vermeiden und mit der Internet Communications Engine (Ice) eine einfache, effiziente Kommunikationsplattform zur Verfügung zu stellen [75]. Im Unterschied zu CORBA handelt es sich um keine Spezifikation, sondern um eine konkrete Implementierung, die eine Unterstützung für C++, .NET, Java, Python, Ruby und PHP bietet und für verschiedene Plattformen zur Verfügung steht (Windows, Linux, MacOS, etc.). Dadurch können Änderungen und Erweiterungen sehr schnell ermöglicht werden, was bei standardisierter Software meist nicht der Fall ist. Ice wurde 2003 von einer kleinen Gruppe engagierter und erfahrener Personen entworfen, zu denen unter anderem Michi Henning gehört, der bis 2002 im OMG-Komitee für CORBA mitgewirkt hat. Namhafte Anwender von Ice sind z. B. der weit verbreitete VoIP-Service Skype, Hewlett-Packard und die Parallel Systems GmbH.

Wie in Abb. 13.3a und 13.3b zu erkennen, wurden viele Konzepte von CORBA übernommen. So muss für verteilte Objekte die Schnittstelle in der Specification Language für Ice (Slice) definiert werden, aus der anschließend der Slice-Compiler die sprachspezifischen Stubs und Skeletons generiert (Abb. 13.3). Slice verfügt zwar über weniger Sprachkonstrukte als die OMG-IDL (z. B. keine inout Parameter), bietet dafür aber eine höhere Flexibilität. Um ein Ice-Objekt erzeugen zu können, muss der Programmierer die Funktionalität des Skeleton realisieren. Auch bei Ice werden die Objekte mit Hilfe der Objekt-Adapter beim Broker (Ice-Core) für verteilte Zugriffe registriert. Im Unterschied zu CORBA-Objekten können Ice-Objekte über eine Vielzahl an Schnittstellen verfügen.



(a) *Broker-Pattern bei Ice* ***(b)*** *Entwicklungsprozess*

Abbildung 13.3: *Ice-Design*

Ein großer Schwachpunkt von CORBA ist das IIOP-Protokoll. Die Enkodierung der Objektreferenzen ist für dieses beispielsweise sehr komplex und verhindert ein effizientes Marshalling und die Verwendung von Speichersharing. Das Ice-Protokoll hingegen ist einfacher, effizienter und bietet unter anderem Datenkompression und Sammelanfragen (batch requests) [75]. Hierzu hat ZeroC

verschiedene Performancemessungen mit Ice 2.1.0 und TAO 1.4, eine der effizientesten CORBA-Implementierungen, durchgeführt und miteinander verglichen. In Abb. 13.4 ist die Zusammenfassung dargestellt. Die Prozentangabe entspricht dem zum Teil sehr deutlichen Geschwindigkeitsvorteil von Ice gegenüber TAO.

	Linux	Windows XP
Latency		
Twoway	145%	77%
Twoway AMI	220%	136%
Oneway	322%	83%
Batched Oneway	1609%	368%
Throughput		
Byte Sequence	-9%	-39%
Struct Sequence	23%	78%

	Linux	Windows XP
Slow Connections		
Document Transfer	138%	
Structured Data	341%	
Event Distribution		
Latency	up to 271%	not measured
Throughput Unbuffered	up to 302%	not measured
Throughput Buffered	up to 588%	not measured

Abbildung 13.4: *Leistungvergleich zwischen Ice und TAO*

Zusammenfassend kann Ice als die konsequente Weiterentwicklung von CORBA angesehen werden, bei der viele Schwachstellen des Standards vermieden wurden [74]. Dem Programmierer wird eine leicht verständliche, einfach zu implementierende und robuste Middleware zur Verfügung gestellt, die neben dem üblichen TCP/IP-Protokoll auch Datagramaufrufe auf Basis des UDP-Protokolls zulässt.

13.2.3 XML-RPC

Bei der Extensible Markup Language Remote Procedure Call (XML-RPC) handelt es sich um eine 1998 von Dave Winer entwickelte Protokollspezifikation, die es verteilter Software erlaubt, entfernte Funktionsaufrufe durchzuführen. Die Kommunikation erfolgt hierbei über das Hypertext Transfer Protocol (HTTP), wobei die Daten mittels Extensible Markup Language (XML) repräsentiert werden. XML-RPC ist sprachunabhängig und sowohl Client als auch Server können in beliebigen Sprachen, wie C, C++, C#, Java oder Ruby implementiert werden [24].

Bei einer Serveranfrage werden die aufzurufende Methode und die entsprechenden Parameter in das XML-Format übertragen und anschließend per HTTP-POST an den Server geschickt (vgl. Abb. 13.5). Dort wird der Body des XML-Dokumentes ausgewertet und der Funktionsaufruf durchgeführt insofern die Funktion beim XML-RPC Server registriert ist. Der Benutzer kann neben primitiven Datentypen wie boolean, int, double und string auch komplexe, aus Primitiven zusammengesetzte, Datentypen verwenden (z. B. mit Hilfe eines XML-RPC::arrays). Nach Bearbeitung liefert der Server dem Client entweder das Ergebnis oder sendet eine XML-Fehlermeldung. Einweganfragen werden vom XML-RPC nicht unterstützt.

Im Vergleich zu CORBA und Ice wird in der Spezifikation keine Schnittstellendefinition vorgegeben, wodurch der Ansatz so einfach wie möglich gehalten wird. Allerdings treten dadurch viele Fehler erst zur Laufzeit auf, da eine Typüberprüfung nicht zur Übersetzungszeit erfolgen kann. Zudem unterstützt XML-RPC keine verteilten Objekte, da sich der Standard lediglich auf das Ausführen entfernter Funktionen beschränkt. Objektreferenzen, wie sie Ice und CORBA mit Hilfe der Objekt-Adapter verwirklichen, sind nicht vorgesehen. C++ Implementierungen realisieren XML-RPC Funktionen durch Funktoren, bei denen die von XML-RPC Kern aufgerufene execute-Methode ausprogrammiert werden muss. Dies führt dazu, dass es anstatt einer Serverklasse eine Vielzahl von unübersichtlichen Funktoren gibt. Bei Übermittlung großer Datenmengen könnten sich die XML-kodierten Daten sowie die Verwendung des HTTP-Protokolls nachteilig auf die Performance auswirken.

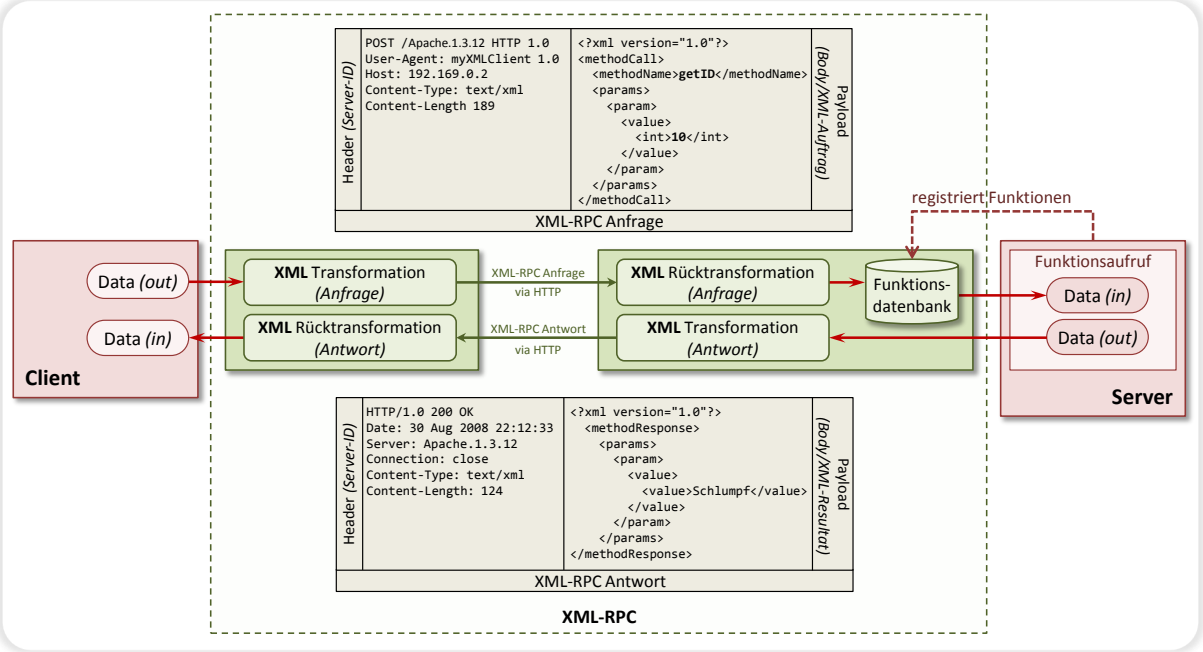


Abbildung 13.5: *Broker-Pattern bei XML-RPC*

13.2.4 SOAP

SOAP wird in vielen Publikationen als Synonym für Webservice verwendet und gilt gemeinhin als Nachfolger bzw. Weiterentwicklung von XML-RPC. Das ursprüngliche Akronym Simple Object Access Protocol wurde mit Erscheinen der Version 1.2 im Juni 2001 fallengelassen, und seither ist SOAP ein eigenständiger, geschützter Name. Bei SOAP handelt es sich um die vom World Wide Web Consortium (W3C) für Webservices veröffentlichte Spezifikation bezüglich des Kommunikationsprotokolls zwischen Client und Server [162]. Diese ist weit verbreitet und wird u. a. von großen Firmen wie Microsoft, Sun und IBM unterstützt. Sie geht über XML-RPC insofern hinaus, als über die Beschränkung auf Wohlgeformtheit der Nachrichten hinaus keine weiteren Einschränkungen hinsichtlich des Komplexitätsgrades für diese bestehen. Somit ist es mit SOAP ohne weiteres möglich, beliebige XML-Daten zu übertragen. Um dies umzusetzen, muss der auf SOAP aufsetzende Webservice definiert, implementiert und veröffentlicht werden (Abb. 13.6).

Genau wie CORBA und Ice verwenden Webservices mit der XML-basierten Web Services Description Language (WSDL) eine eigene Schnittstellendefinitonssprache. WSDL ist eine anbieter- und plattformunabhängige Beschreibung von Webservices, die alle öffentlich verfügbaren Funktionen des jeweiligen Interfaces, die notwendigen Datentypinformationen, das zu verwendende Protokoll (z. B. TCP, HTTP, SMTP, Jabber) sowie die Adresse, über die der Service zu erreichen ist, beschreibt. Erfüllt die Anfrage eines Clients nicht die WSDL-Richtlinien des Services, so erhält dieser eine entsprechende SOAP-Fehlermeldung. Um Webservices zur Verfügung zu stellen bzw. um die benötigten WSDL-Daten zu beziehen, kommen meist globale UDDI(Universal Description, Discovery and Integration)-Verzeichnisdienste zum Einsatz [15].

In Abb. 13.7 ist exemplarisch die Entwicklung und Anwendung einer Client-Server-Anwendung auf Basis von gSOAP [34], einer sehr verbreiteten und effizienten auf C/C++-basierenden SOAP-Implementierung, dargestellt. Auch hier besteht große Ähnlichkeit zu der Vorgehensweise bei CORBA und Ice. Die Proxys konvertieren bei ausgehenden Anfragen bzw. Antworten die Objekte

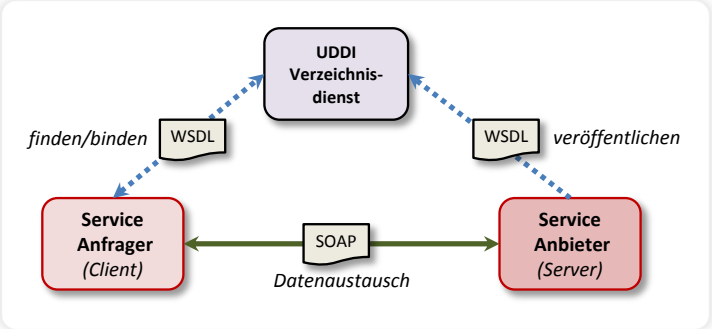


Abbildung 13.6: *Funktionsweise eines Webservices*

und Strukturen der verwendeten Programmiersprachen in die SOAP-Datenstruktur und erzeugen Remote-Calls, die die Methodenaufrufe der jeweiligen Objekte der höheren Programmiersprachen in ihren Signaturen abbilden. Die einfachen Datentypen werden den entsprechenden XML-Datentypen zugeordnet. Auf der Empfängerseite erfolgt analog die Rücktransformation und der Funktionsaufruf. Auch hier sind aufgrund der Standardisierung die Implementierungen unterschiedlicher Programmiersprachen zueinander kompatibel. Die Analyse der XML-Daten führt jedoch zu erhöhter CPU-Last, höherer Latenzzeiten und reduziert aufgrund des Datenüberschusses innerhalb des XML-Dokuments den Datendurchsatz. Dies führt im Vergleich mit CORBA und Ice zu sehr schlechten Performanceleistungen [25, 58, 33]. Wie XML-RPC unterstützen auch Webservices nicht die Behandlung verteilter Objekte.

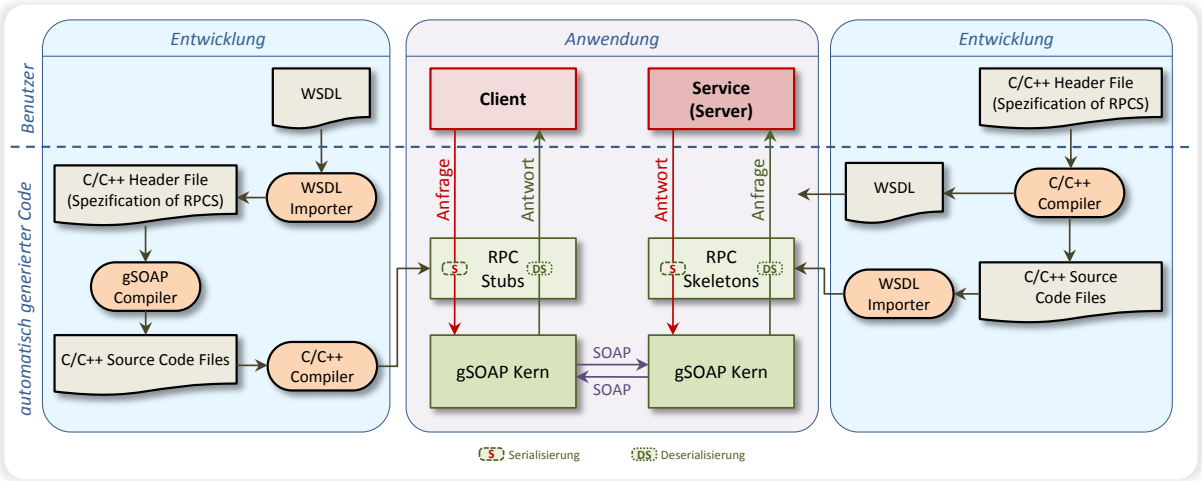


Abbildung 13.7: *Entwicklung und Anwendung von SOAP am Beispiel von gSOAP*

13.2.5 RCF

Das Remote Call Framework (RCF) von Jarl Lindrud ist, wie Ice, keine Spezifikation, sondern eine konkrete Implementierung. Das ursprünglich 2005 über das CodeProject-Portal [20] veröffentlichte Projekt wird derzeit von Jarl Lindrud für eine australische Firma weiter entwickelt, um dort Microsofts DCOM zu ersetzen. Der Öffentlichkeit steht RCF unter der GNU General Public License v2 zur Ver-

fügung. Anders als die bisher vorgestellten Konzepte ist RCF zwar plattformunabhängig, aber nicht sprachunabhängig und steht ausschließlich für C++-Anwendungen zur Verfügung.

Die verteilte Funktionalität wird ebenfalls über Schnittstellendefinition ermöglicht. Im Gegensatz zu CORBA und Ice werden Schnittstellen- und Typdefinitionen dabei nicht über ein aufwändiges Objekt-Modell repräsentiert. Aufgrund der Sprachbindung erfolgen die Definitionen direkt in Standard-C++ und benötigen keinen separaten IDL-Compiler (vgl. Abb. 13.8). Dadurch stellt RCF ein einfacheres und verständlicheres Programmiermodell zur Verfügung als die bisher vorgestellten, sprachunabhängigen Konzepte.

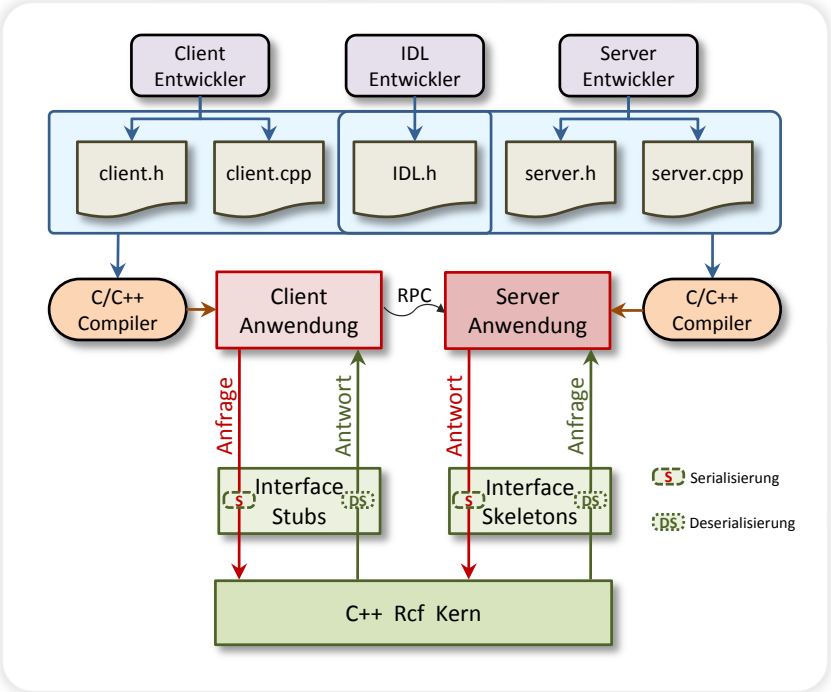


Abbildung 13.8: *Client/Server-Entwicklung mit RCF*

Um das Marshalling zu ermöglichen, müssen die zu übertragenden Objekte zuvor serialisiert, also in eine Bytesequenz konvertiert werden. Anders als in modernen, objektorientierten Sprachen wie Java oder C# sieht der C++-Standard keine automatische Serialisierung vor. Um dem zu begegnen, stellt RCF mit dem Serialization Framework (SF) diese Funktionalität für diverse Typen, wie elementare Datentypen (integer, double, etc.) und STL-Container (std::vector<>, std::map<>, etc.), zur Verfügung. Selbst definierte Klassen müssen hierfür eine Serialisierungsmethode zur Verfügung stellen. In dieser kann der Entwickler auch ältere Versionen der Klasse berücksichtigen und somit letztlich Kompatibilität zwischen Client und Server mit verschiedenen Versionen berücksichtigen. SF verwendet hier bewusst die Syntax der Serialisierungsbibliothek von Boost [8], einer der am weitesten verbreiteten C++-Bibliotheken, die dadurch alternativ zu SF verwendet werden kann. In der Praxis hat sich jedoch gezeigt, dass das performantere SF bevorzugt verwendet werden sollte. Das serialisierte Objekt kann mit dem gewählten Transportprotokoll (TCP/IP oder UDP) versendet werden. Optional können ein Datenkompressionsfilter hinzu geschaltet werden und/oder eigene Protokolle/Filter verwendet werden.

Um Objekte für RPCs zur Verfügung zu stellen, muss die zugehörige Klasse die im Interface definierte Funktionalität (=Methoden) implementieren. Anschließend kann die Schnittstelle zur Laufzeit an das jeweilige Objekt gebunden und so über einen Server verfügbar gemacht werden. Anders als bei den

anderen Ansätzen bedarf die Bereitstellung dieser Methoden keiner Ableitung einer RCF-spezifischen Klasse. Dies gewährleistet, dass die Klassen auch außerhalb des RCF-Kontextes verwendet werden können.

Über die Basisfunktionalität hinaus bietet RCF u. a. automatische Speicherverwaltung mittels intelligenter Zeiger (boost::shared_ptr), Fehlerbehandlung mittels C++-Ausnahmen (exceptions), Threadsicherheit, Publish/Subscribe-Dienste, Unterstützung von Multicast- und Broadcastaufrufen, Remote Object Creation, verschlüsselte Datenübertragung (Kerberos, NTLM, SSL) sowie die Verwendung von Callback-Funktionen. Serverseitig können gleichzeitig mehrere Objekte mit dem gleichen Interface zur Verfügung gestellt werden. Diese Möglichkeit der verteilten Objektbehandlung grenzt RCF deutlich von XML-RPC und SOAP ab.

Ein Nachteil, neben der Sprachbindung, ist, dass RCF von lediglich einer Person entwickelt wird und keiner Spezifikation unterliegt. In naher Zukunft soll RCF jedoch kommerzialisiert werden.

13.3 Fazit

MPI ist in Hinblick auf einen parallelen und adaptiven Strömungslöser mit dynamischer Kommunikation nur bedingt geeignet. Eine Umsetzung ist zwar theoretisch möglich, bedarf aber eines verhältnismäßig hohen Implementierungsaufwands und garantiert keine Plattformunabhängigkeit. MPI wurde für eine starre aber effektive Nachrichtenkommunikation zwischen verteilten Prozessen konzipiert und ist dort nach wie vor das Mittel der Wahl.

Im Gegensatz dazu bieten CORBA und Ice mit ihren Konzepten die größte Funktionalität. CORBA hat jedoch einige Mängel, wie die hohe, oftmals unnötige Komplexität, und scheint durch die fehlende Unterstützung des .NET-Frameworks nicht mehr auf der Höhe der Zeit zu sein. Ice hingegen repräsentiert sich als konsequente Neuentwicklung von CORBA, bei der die guten Konzepte übernommen und die Designfehler beseitigt wurden. Dadurch steht dem Softwareentwickler ein wesentlich einfacher zu handhabendes, effizientes Framework zur Verfügung.

SOAP und XML-RPC stellen die beste Lösung für Webapplikationen dar, bei denen die Performance zweitrangig ist. SOAP bietet hier die größtmögliche Kompatibilität, insbesondere hinsichtlich Firewalls, die bei CORBA und Ice oftmals Probleme bereitet.

Die Vielzahl der möglichen Implementierungssprachen bedeutet einen weiteren Nachteil: Ein Standard muss sich auf die gemeinsamen Eigenschaften dieser Sprachen beschränken. Bezogen auf VIRTUALFLUIDS stellt RCF deshalb die beste Lösung dar. Die Sprachunabhängigkeit wird nicht benötigt und durch den Wegfall der spezifischen IDL ist die Integration in das bestehende Paket mit relativ geringem Aufwand möglich. Andernfalls müsste für jede zu übertragende Klasse das entsprechende IDL-Pendant erzeugt, implementiert und ggf. zusätzliche Typkonvertierungen durchgeführt werden, falls der benötigte Typ nicht von der Middleware unterstützt wird. Änderungen der Klassendeklarationen oder Funktionsparameter wären dadurch unnötig aufwändiger. Zudem fällt die Einarbeitung in eine spezielle Schnittstellensprache weg.

Ein oft genannter Nachteil von Middleware ist deren Größe und Schwerfälligkeit sowie die fehlende Möglichkeit der Leistungsoptimierung durch den Programmierer. Dies trifft auf RCF nur bedingt zu, da der Code sehr übersichtlich und gut dokumentiert ist. Zudem bestand während der Entwicklung von VIRTUALFLUIDS enge Zusammenarbeit mit dem Entwickler Jarl Lindrud.

14 Hybride Blockgitter-Erweiterung

Eine vielversprechende Alternative zum Knotengitter ist die Verwendung blockstrukturierter Gitter. Diese kombinieren die Vorteile der hierarchischen mit denen der kartesischen Gitterstruktur. Man spricht in diesem Zusammenhang daher auch oft von *hybriden Blockgittern*. Ein Anwendungsbeispiel ist in Abb. 14.1 in Form einer Blockgitterverfeinerung eines Kühlturms dargestellt:

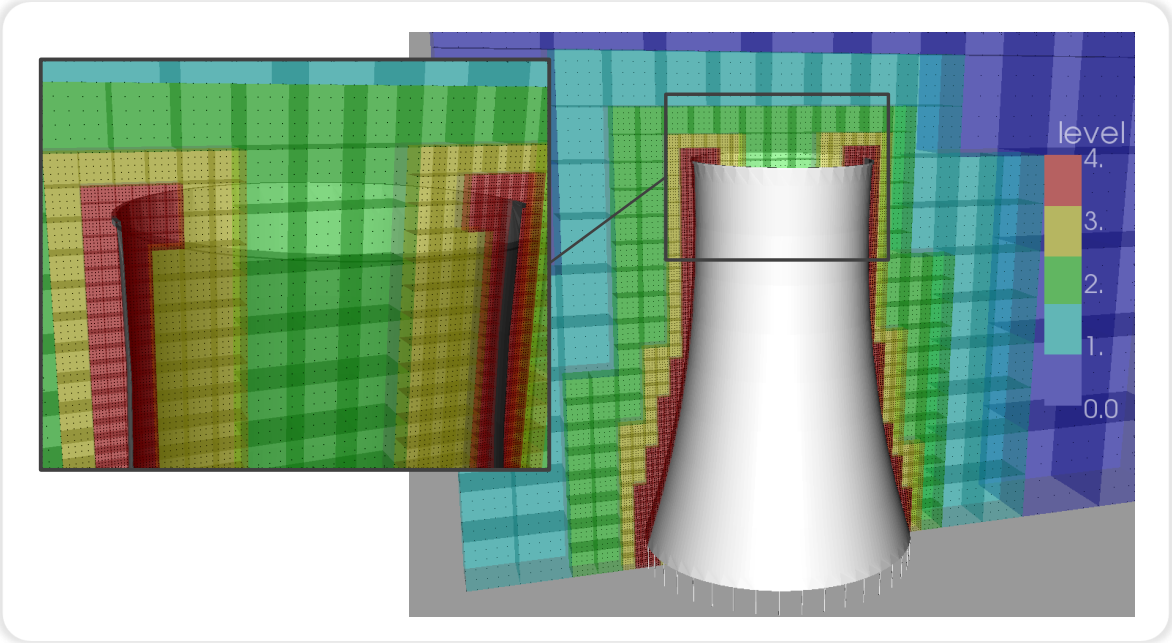


Abbildung 14.1: *Kühlturm mit hybrider Blockgitterverfeinerung*

Zunächst wird das Strömungsgebiet mit Hilfe der *Block-Mesh-Refinement(BMR)*-Technik in diskrete Blöcke, z. B. auf Basis eines Quad-/Octtrees, aufgeteilt. In diesen werden anschließend die numerischen Teilgitter für die eigentliche Simulation initialisiert, z. B. in Form von gleich dimensionierten Knotenmatrizen pro Block.

Diese Struktur hat viele Vorteile. So ist durch die geringere Anzahl von Blöcken im Vergleich zu den Knoten eine schnellere Gebietsdiskretisierung sowie -zerlegung möglich, was insbesondere bei adaptiven Berechnungen zur Geltung kommt. Die Verwendung von uniformen Knotengittern in Form von Matrizen innerhalb der Blöcke ermöglicht die Verwendung effizienter Algorithmen und benötigt durch die Möglichkeit der direkten Adressierung weniger Rechenressourcen. Beschränkt man die Interblockkommunikation auf die Blockränder/-flächen, begünstigt dies die Parallelisierung. Denkbar ist auch die Verwendung unterschiedlicher numerischer Rechenkerne für verschiedene Blöcke.

Die gröbere Gebietsdiskretisierung („Legostruktur“) im Vergleich zu reinen Knotengittern kann hier als nachteilig angesehen werden. Da die physikalische Blockgröße durch die interne Matrixauflösung

bestimmt wird, die wiederum maßgeblich für die Berechnungseffizienz ist, gilt es, das jeweilige Optimum zu finden. Auch könnte eine unvermeidbare unausgewogene Blockverteilung bei verteilten Berechnungen mit geringer Block- und hoher Prozessanzahl in einer ungünstigeren Lastverteilung resultieren.

In VIRTUALFLUIDS konnte bei der Implementierung des Blockgitters im Wesentlichen sowohl die Datenstruktur als auch die Einteilung in Topologie-, Gitter- und Physikschicht vom Knotengitter übernommen werden (Abb. 14.2).

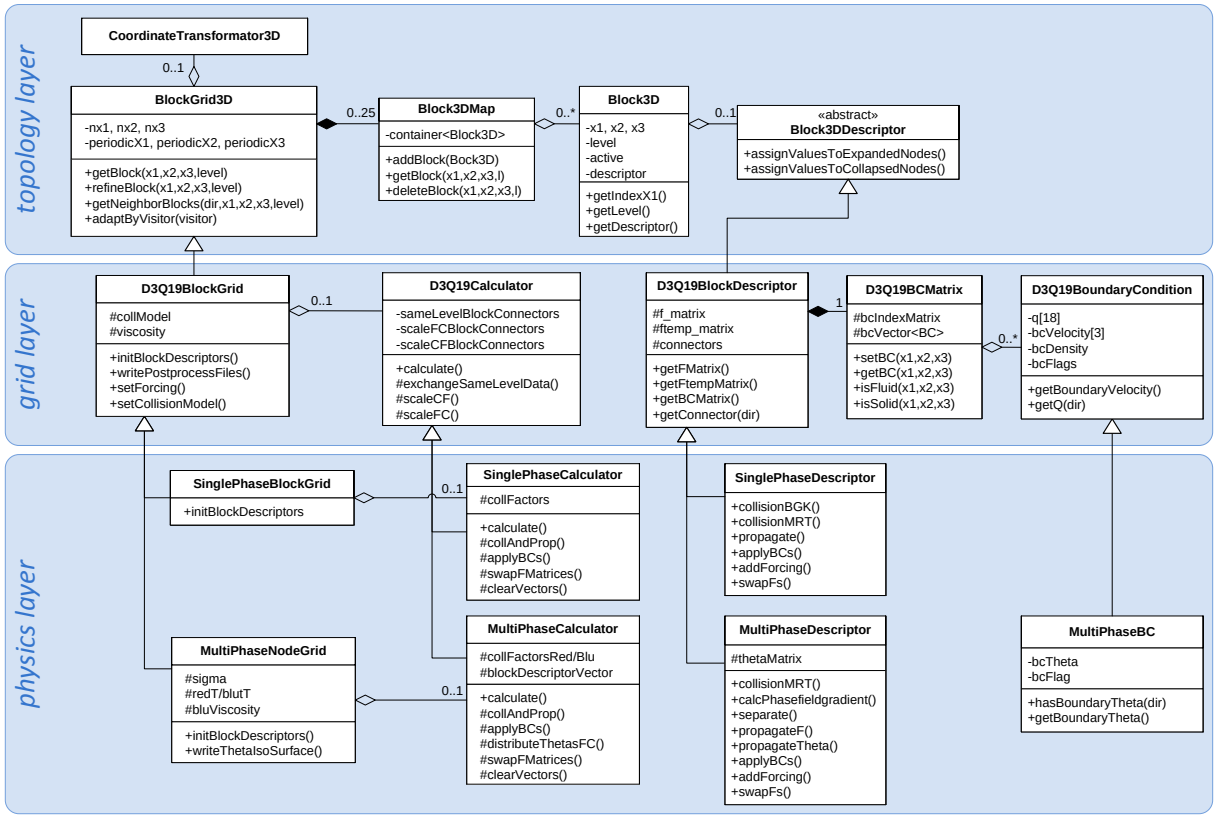


Abbildung 14.2: *Blockgitter (UML)*

Die Knoten wurden durch Blöcke substituiert und die topologischen Algorithmen, wie die Nachbarermittlung und die Blockvergröberung/-verfeinerung, entsprechend angepasst. Ein Block kann bei einem geglättetem Gitter im Gegensatz zu einem Knoten im Knotengitter beispielsweise bis zu vier Nachbarblöcke für eine Himmelsrichtung besitzen. Die Indizierung der Blöcke erfolgt nach dem im Abschn. 6.1 beschriebenen Schema.

Um in der Berechnung effiziente, matrixbasierte Algorithmen anwenden zu können, wurde in den speziellen Blockdeskriptoren auf Knotenobjekte verzichtet. Stattdessen speichern diese die LB-Knotenverteilungen in Form von mehrdimensionalen Feldern (Abb. 14.3). Die Kollisions- und Propagationsroutinen wurden aus der Berechnungsklasse in die Deskriptoren verschoben. Durch die Verlagerung der Verantwortlichkeit kann eine Berechnungsklasse verschiedene Arten von Deskriptoren unterstützen. Auch können so mit geringem Aufwand Optimierungen durchgeführt werden, wie z. B. die Verwendung dünnbesetzter Matrizen (*engl.:* sparse matrices) bei teilgefüllten (Rand-)Blöcken.

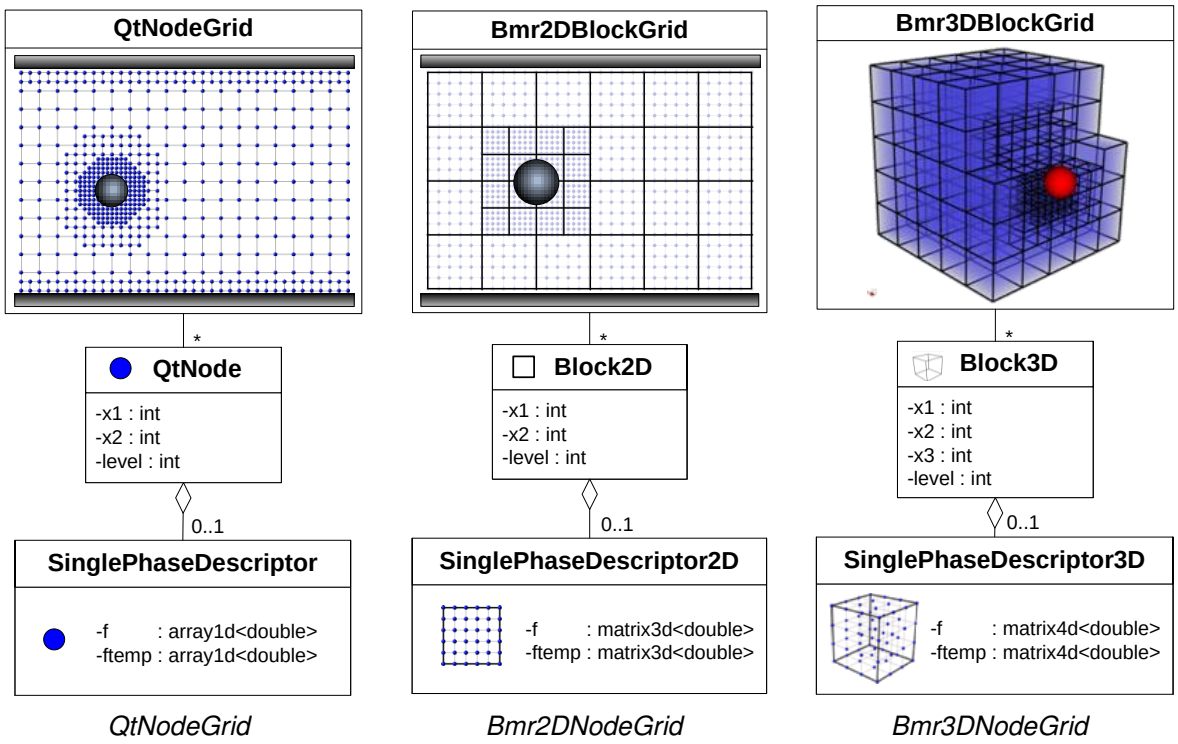


Abbildung 14.3: *Analogie der Knoten- und Blockdatenstruktur*

Zur Knotentypisierung verwendet der Deskriptor die D3Q19BCMatrix. Diese Matrix speichert die Information, ob es sich bei einem Knoten um einen Fluid-, Festkörper- oder Randbedingungsknoten handelt. Für letztere werden zusätzlich D3Q19BoundaryCondition-Randbedingungsobjekte vorgehalten.

Analog zum Knotengitter werden für LB-Simulationen auch beim Blockansatz geglättete Gitter verwendet, die mittels Block- und Gitterbesucherklassen erzeugt werden (vgl. Abschn. 6.3). Die Levelweite kann hier vernachlässigt werden, da ein Block in der Regel über eine ausreichende Anzahl an Knoten pro Richtung verfügt. Im Allgemeinen werden Gitterbesucherklassen hauptsächlich für Gesamtgitteroperationen verwendet, wie das Anpassen der Viskosität oder das Erzeugen von Isoflächen. Die Algorithmen der Blockbesucherklassen hingegen verändern (wahlweise iterativ) die Eigenschaften einzelner Blöcke, wie durch die Initialisierung der Knotenverteilungen oder das Verfeinern von Blöcken.

Eine Übersicht über die wesentlichen Klassen des Blockgitters ist in Abschn. A4 zu finden. Auf die Details wird in den folgenden Kapiteln eingegangen.

15 Interblockkommunikation (Connector-Transmitter-Konzept)

Bei der Umsetzung der Blockstruktur wurde in Hinsicht auf zukünftige Erweiterungen (z. B. unterschiedliche numerische Rechenkerne pro Block) festgelegt, dass die Blöcke während eines Berechnungsschrittes autark, d. h. ohne Zugriffe auf Nachbarblöcke, arbeiten. Diese Unabhängigkeit vereinfacht die Interblockkommunikation und die Programmparallelisierung. Für den Datenaustausch über die Blockgrenzen hinweg wurde das *Connector-Transmitter-Konzept* entwickelt (Abb. 15.1). Die strikte Trennung von Connectoren und Transmittern hat den Vorteil, dass Quellcoderedundanz weitestgehend vermieden wird und lediglich der Transmittertyp darüber entscheidet, ob es sich um einen prozessübergreifenden Datenaustausch oder einen lokalen handelt. Um größtmögliche Flexibilität und Effizienz zu erreichen, kommt hier generische Programmierung unter Verwendung der in C++ zur Verfügung stehenden Templateprogrammierung zum Einsatz [5, 161].

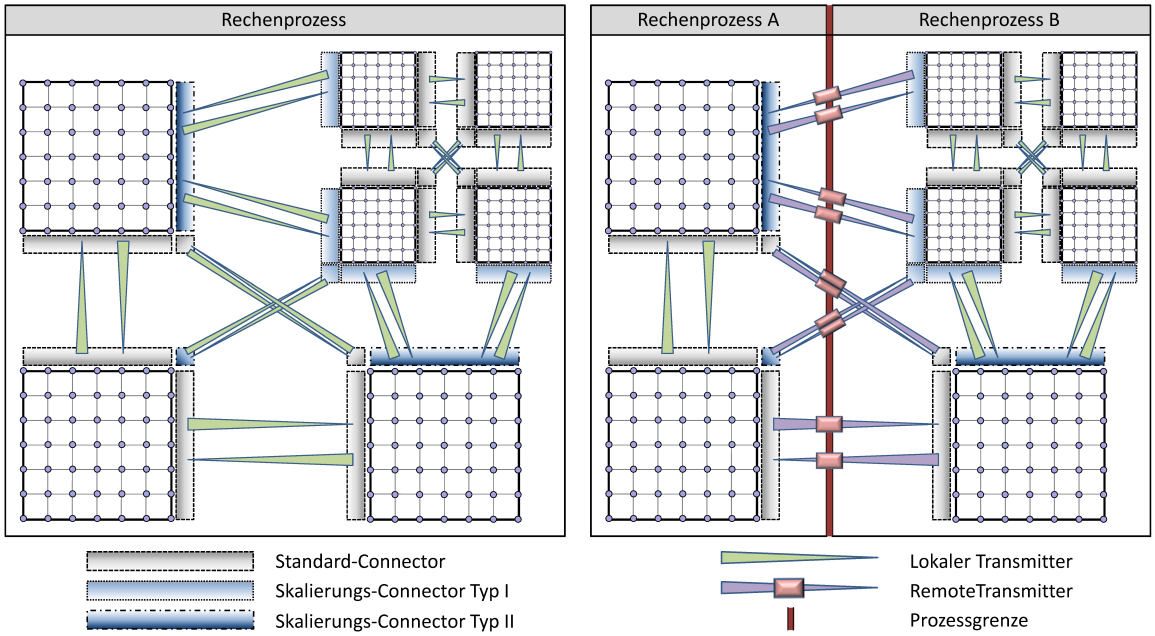


Abbildung 15.1: *Connector-Transmitter-Konzept für lokale und verteilte Berechnungen*

15.1 Transmitter

Transmitter-Module können als Sender, Empfänger oder beides fungieren. Sie kapseln den eigentlichen Datentransport von einem Ort zum anderen. Je nach Implementierung kann es sich um eine lokale oder eine prozessübergreifende Übermittlung handeln.

Die Transmitter enthalten die zu übertragenden Daten (z. B. Datenvektoren). Zudem stellen sie verschiedene Schnittstellen zur Verfügung über die das Verhalten der Transmitter definiert wird. Es existieren u. a. Methoden für die Datenpufferinitialisierung sowie -synchronisierung der Sende- und Empfangseinheit im Präprozess und während der Berechnungslaufzeit.

In Abb. 15.2 sind die derzeit in VIRTUALFLUIDS zur Verfügung stehenden Transmitterklassen abgebildet. Mit Ausnahme der LocalTransmitter-Klasse erfolgt die Übertragung der Daten vom Sender zum passenden Empfänger mittels physikalischer Kopie. Die verteilt arbeitenden Klassen (RemoteTransmitter) unterscheiden sich vor allem hinsichtlich der verwendeten Interprozessbibliothek (MPI, RCF, MUSCLE) und der Art der Übermittlung (blockierend/nicht blockierend). Auf die Besonderheit der Pooltransmitter wird in Abs. 17.8.3 eingegangen.

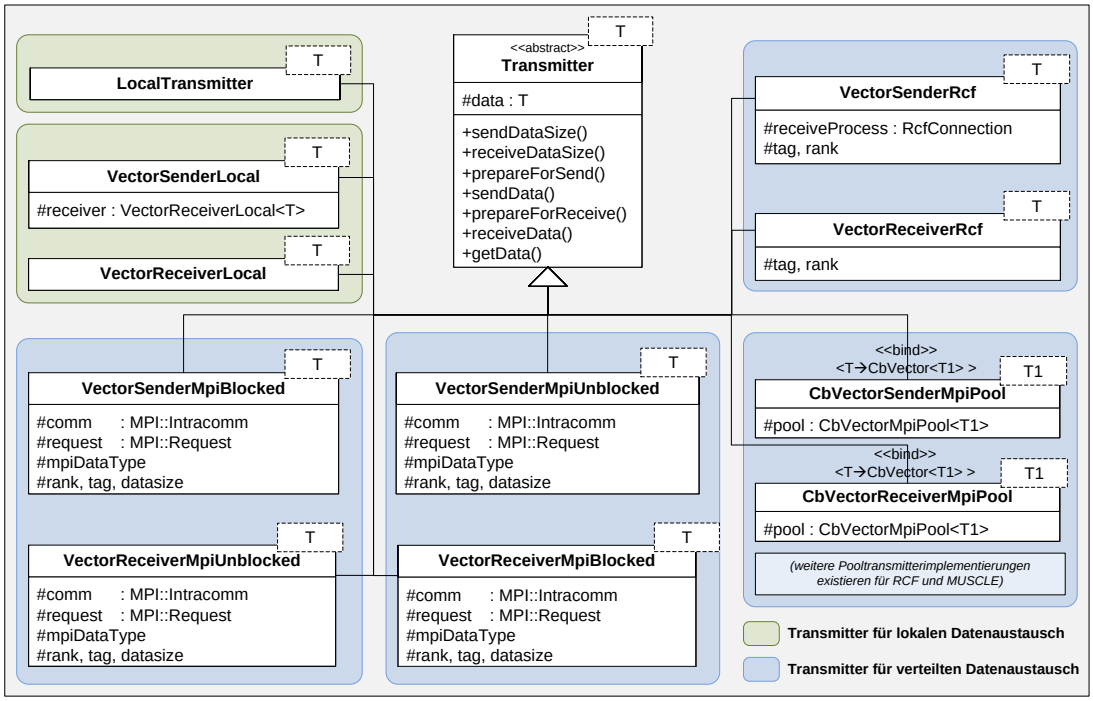


Abbildung 15.2: *Transmitterklassen von* VIRTUALFLUIDS

15.2 Connectoren

Connectoren sind Verbindungsmodule und beinhalten in Abhängigkeit der jeweiligen Diskretisierungsrichtung die Logik, die für einen Nachbarblock benötigten Daten einzusammeln bzw. die vom Nachbarblock empfangenen Daten einem Block zuzuweisen. Als Datencontainer können hier *Transmitter* zum Einsatz kommen.

Hinsichtlich der Datenübertagung wird zwischen zwei Arten von Connectoren unterschieden:

- **DirectConnector**

  Der *DirectConnector* verfügt sowohl über den Quell- als auch über den Zielblockdeskriptor. Dadurch kann er auf beide Blöcke zugreifen und die betreffenden Informationen *direkt* austauschen.

Ein Beispiel für diesen Typ ist der D3Q19SameLevelDirectConnector. Dieser kopiert die benötigten Verteilungen der betreffenden Blockkante bzw. -seitenfläche durch direkten Zugriff auf die Verteilungsmatrizen vom Quell- zum Zieldeskriptor (Abb. 15.3).

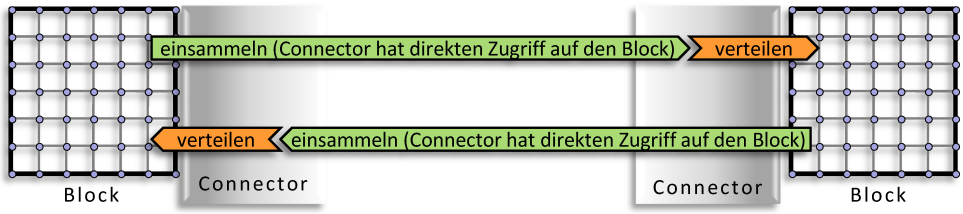


Abbildung 15.3: *DirectConnector*

- **VectorConnector**

  Im Gegensatz zum *DirectConnector* hat der *VectorConnector* nur Zugriff auf einen Blockdeskriptor. Die zu übertragenden Informationen werden in einem Datenvektor zwischengespeichert, der anschließend mittels Transmitter zum Nachbarconnector übertragen wird. Dort werden die Daten aus dem Vektor in den betreffenden Matrizen einsortiert. Ein Beispiel hierfür ist der D3Q19SameLevelVectorConnector. Befinden sich beide Blöcke auf demselben Prozess, können sich die Connectoren einen Transmitter teilen (z. B. LocalTransmitter). Auf der einen Seite fungiert er in diesem Fall als Sender und auf der anderen Seite als Empfänger (Abb. 15.4). RemoteTransmitter kommen zum Einsatz, wenn sich ein Nachbarblock auf einem anderen Prozess befindet (Abb. 15.5).

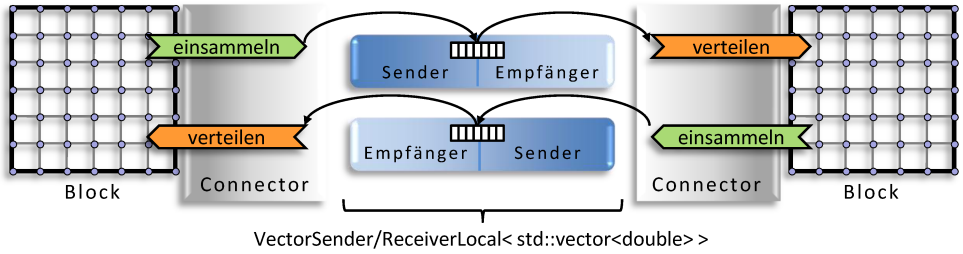


Abbildung 15.4: *VectorConnector mit lokalem EinvektorTransmittern*

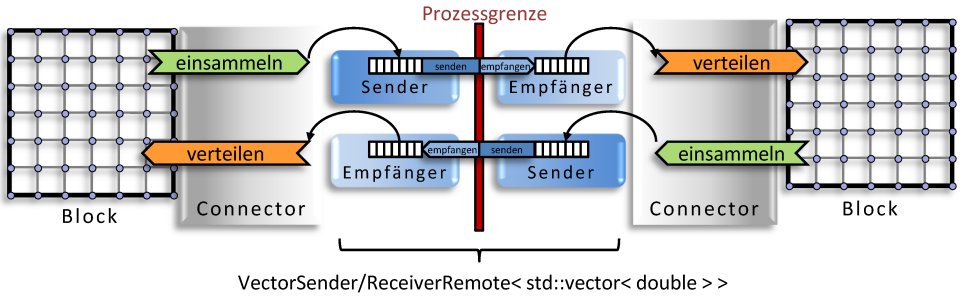


Abbildung 15.5: *VectorConnector mit RemoteTransmittern*

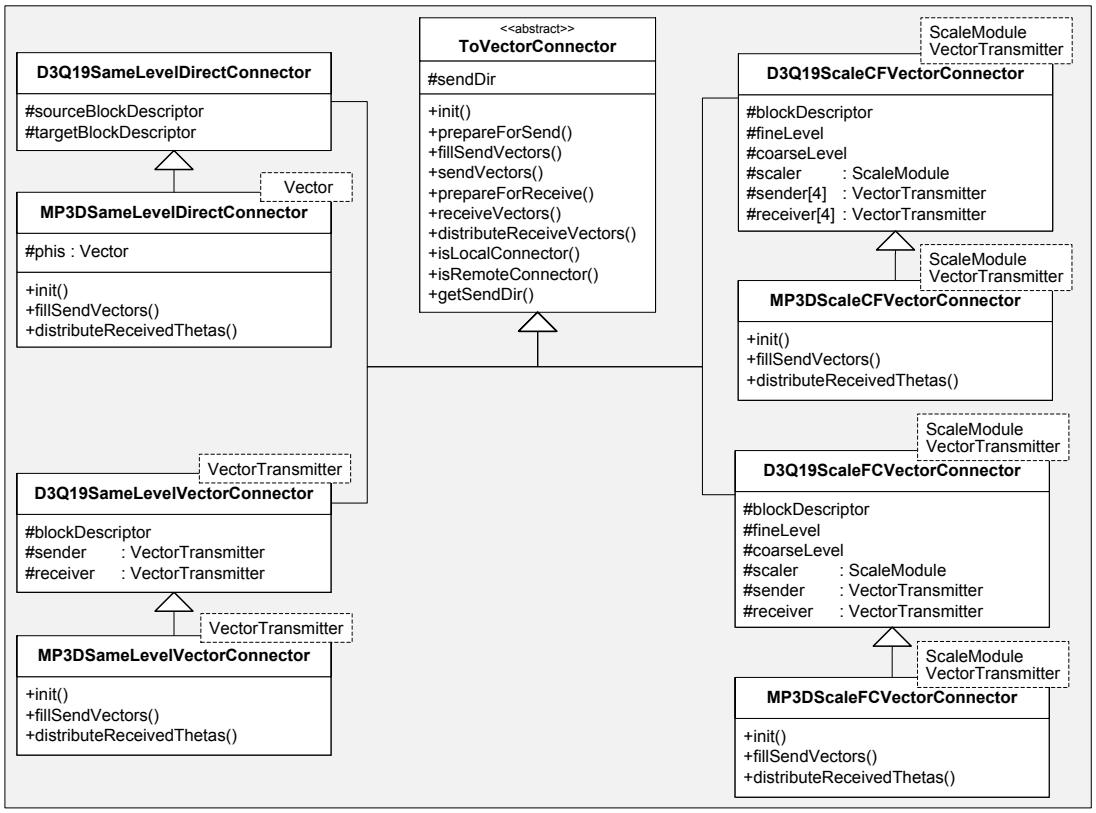


Abbildung 15.6: *Connectoren für Ein- und Mehrphasenprobleme (d3q19-Modell)*

VIRTUALFLUIDS verwendet die in Abb. 15.6 abgebildeten Connectoren. Neben der Art der Datenübertragung wird dort zudem anhand der vorhandenen Gitterleveldifferenz der zu verbindenden Blöcke zwischen folgenden Typen unterschieden:

- **SameLevelConnector**

  Diese verbinden zwei Blöcke gleichen Levels miteinander und sammeln bzw. verteilen, wie bereits beschrieben, die benötigten Informationen.

- **ScaleConnector**

  Der Skalierungsconnector realisiert die Kommunikation zwischen Blöcken unterschiedlicher Gitterebenen. Im Gegensatz zum SameLevelConnector können hier die Daten nicht einfach kopiert werden, denn, wie in Abschn. 3.3 beschrieben, müssen die Informationen zwischen unterschiedlichen Auflösungsstufen skaliert, raum- und zeitinterpoliert werden. Die *ScaleConnectoren*, die diese komplexe Aufgabe übernehmen, werden anhand der Skalierungsrichtung unterschieden (fein→grob: FC bzw. grob→fein: CF). Hierbei kommuniziert ein *ScaleFCConnector* mit **genau einem** *ScaleCFConnector*, während ein *ScaleCFConnector* je nach Richtung mit bis zu zwei (2-D) bzw. vier (3-D) *ScaleFCConnectoren* Daten austauscht (Abb. 15.1).

  Für die Skalierung werden als weitere Abstrahierung Skalierungsmodule verwendet:

Das *Skalierungsmodul (ScaleModule)* ist eine austauschbare Hilfsklasse des *ScaleConnectors*, die den Prozess der Knotenskalierung in Abhängigkeit des verwendeten LB-Modells durchführt.

Dadurch lässt sich ein *ScaleConnector* für eine Vielzahl von LB-Modellen verwenden (z. B. in-/kompressibles Momenten-/BGK-Modell für Ein-/Mehrphase, Abb. 15.7).

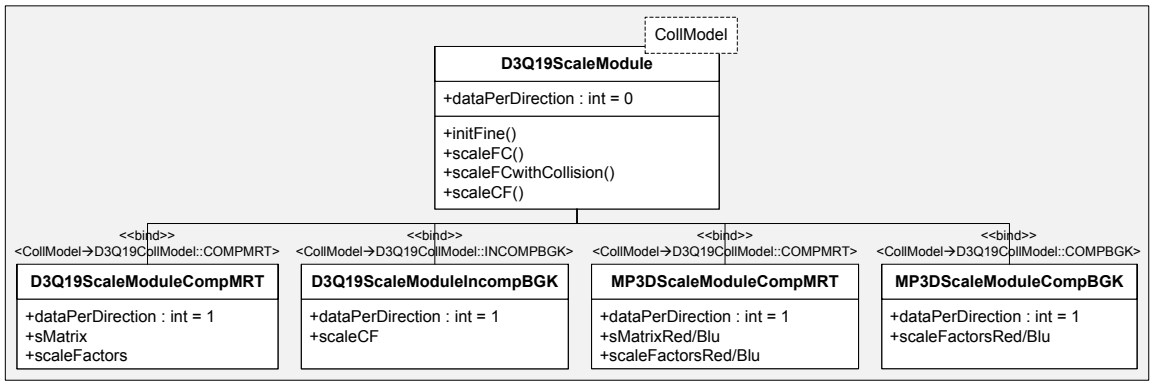


Abbildung 15.7: *Skalierungsmodule für Ein- und Mehrphasenprobleme (d3q19-Modell)*

Um den notwendigen Transfer der Phasenfeldinformationen bei Mehrphasenberechnungen zu berücksichtigen, wird diese zusätzliche Funktionalität durch Vererbung in den entsprechenden Mehrphasenconnectoren zur Verfügung gestellt (Abb. 15.7). Die Methoden für das Strömungsfeld werden übernommen, und die Zusatzinformationen werden bei den VectorConnectoren hinter die vorhandenen Daten des Transmitterobjekts angefügt. Bei MP3DScaleConnectoren kommen spezielle Mehrphasenskalierungsmodule zum Einsatz.

Das hier vorgestellte Connector-Transmitter-Konzept bietet zudem die Basis, Blöcke unterschiedlicher numerische Kernen zu koppeln. Theoretisch sind auch Kopplungen zwischen 2-D- (z. B. Flachwasser) und 3-D-Blöcken (z. B. freie Oberfläche) denkbar. Hierzu müssen die jeweiligen Verbindungsmodule für die benötigte Übertragung der physikalischen Größen definiert und implementiert werden.

16 Interaktoren

Um Festkörper zu berücksichtigen, kommen in VIRTUALFLUIDS Interaktorklassen zum Einsatz. Diese verbinden das Blockgitter mit der eigentlichen Geometrie. Neben der Abbildung auf das Blockgitter werden diese auch zur Gitterverfeinerung entlang der Geometrieoberfläche, zur Kraftauswertung und mit Hilfe von Bmr3DSteeringPlugins zur Bewegungssteuerung der Geometrieobjekte verwendet (Abb. 16.1).

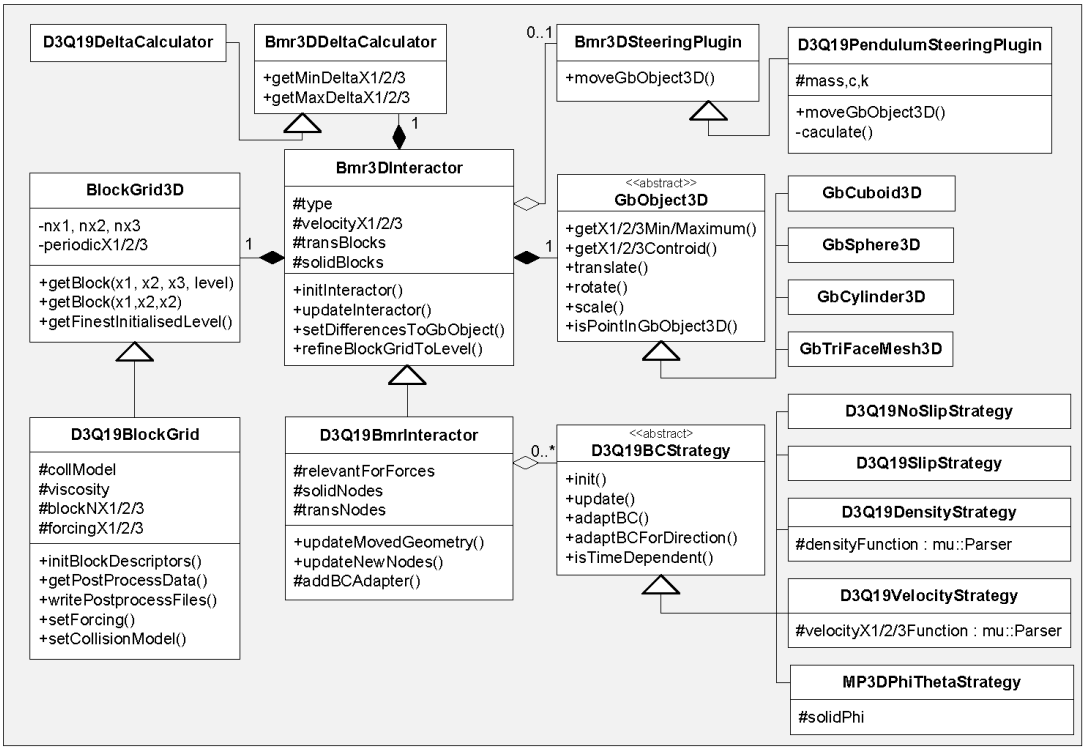


Abbildung 16.1: *Übersicht der interaktorspezifischen Klassen*

16.1 Standardinteraktor

Im Folgenden wird der Standardalgorithmus zur Abbildung und Initialisierung primitiver Festkörper beschrieben. Von der Basisklasse Bmr3DInteractor werden Geometrieobjekte unterstützt, die über entsprechende *Punkt-in-Objekt-Tests* verfügen. Der davon abgeleitete D3Q19BmrInteractor benötigt zusätzlich noch *Punkt-Abstands-Berechnungsmethoden* auf Basis von Raytracing- oder ClipLine-Methoden. Die Diskretisierung der Geometrie auf das Blockgitter erfolgt in zwei Schritten, die im Folgenden im Detail erläutert werden.

16.1.1 Diskretisierung der Blöcke

Als erstes wird der umhüllende Quader (BoundingBox) der jeweiligen Geometrie bestimmt, mit der anschließend für jeden Gitterlevel die vorhandenen Blöcke verschnitten werden. Sowohl die BoundingBox als auch die Blöcke werden dabei durch eine Axis Aligned Bounding Box (AABB) repräsentiert, für die eine Vielzahl performanter Verschneidungsalgorithmen existiert [4, 141]. Im Allgemeinen ist dieser Test effizienter als die bei positiver Verschneidung anknüpfende Verschneidung mit der eigentlichen Geometrie.

Um ein von den tatsächlichen geometrischen Abmessungen eines Blocks abweichendes Einflussgebiet zu berücksichtigen, können die Blockdimensionen für den Verschneidungstest individuell vom Anwender angepasst werden. Dies wird u. a. bei der Bestimmung der für die Boundary-Fitting-Methode erforderlichen normierten Knotenabstände **q** benötigt (vgl. Abschn. 3.2):

$$q_i = \frac{Abstand_{Knoten \rightarrow Geometrie\ in\ Richtung\ \mathbf{e}_i}}{|\mathbf{e}_i\ \Delta x_{Level}|} \quad \text{mit} \quad q_i \in]0,\ldots,1] \tag{16.1}$$

Somit muss die Blockgeometrie für den Verschneidungstest entsprechend um Δx_{Level} vergrößert werden, damit die Knoten auf dem Blockrand entsprechend berücksichtigt werden können (*angepasster Block*, Abb. 16.2). Hierfür verwendet der Interaktor den D3Q19DeltaCalculator, der levelweise die zu berücksichtigenden Überlängen vorhält.

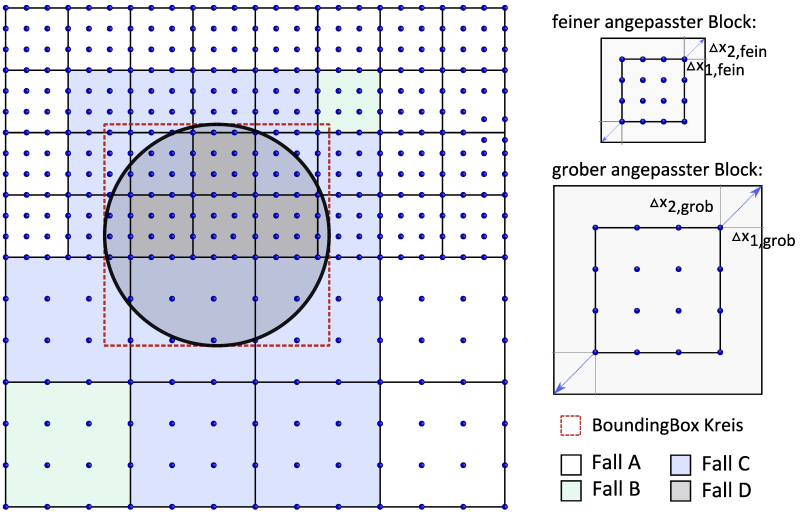


Abbildung 16.2: *Projektion eines Festkörpers (Kreis) auf die Blöcke des Blockgitters*

Wie in Abb. 16.2 dargestellt, können bei der Verschneidung von Block und Festkörpergeometrie folgende Fälle eintreten:

Fall A angepasster Block liegt komplett außerhalb der BoundingBox
→ *keine weiteren Tests*

Fall B angepasster Block schneidet oder liegt vollständig innerhalb der BoundingBox
angepasster Block hat keine gemeinsame Schnittfläche mit dem Geometrieobjekt
→ *keine weiteren Tests*

Fall C angepasster Block schneidet oder liegt vollständig innerhalb der BoundingBox
angepasster Block schneidet Umrandung der Geometrie
→ *Block wird als Randblock gespeichert und dessen Knoten werden diskretisiert*

Fall D angepasster Block liegt vollständig innerhalb BoundingBox
nicht angepasster Block vollständig innerhalb der Geometrie
→ *Block wird als Festkörperblock gespeichert und solid gesetzt*

16.1.2 Diskretisierung der Knoten

Im D3Q19BmrInteractor werden alle Knoten der Blöcke behandelt, die eine Verschneidung mit der Geometrie aufweisen, aber nicht *solid* sind (Fall C, Abb. 16.2). Jeder Knoten innerhalb einer Geometrie wird analog zum Block zu einem *solid* Knoten und hinterher in der Berechnung nicht berücksichtigt. Jeder Fluidknoten der sich unmittelbar am Rand der Geometrie befindet, erhält in Abhängigkeit des Geometrietyps eine D3Q19BoundaryCondition. Die restlichen Knoten bleiben unverändert.

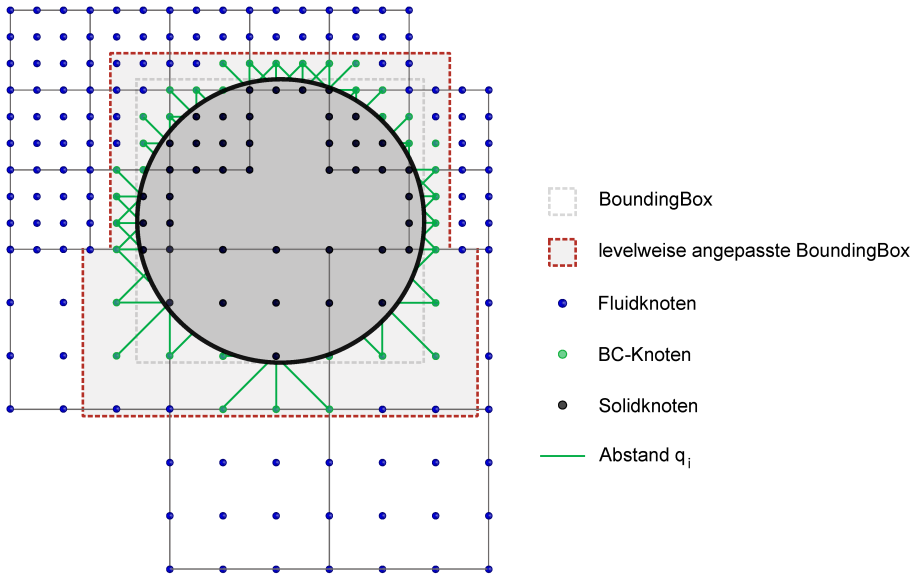


Abbildung 16.3: *Projektion eines Festkörpers (Kreis) auf die Knoten der Blöcke*

Zu Beginn wird überprüft, ob sich ein Knoten innerhalb der BoundingBox des Festkörpers befindet. Diese wird, um den Einflussbereich der Knoten zu berücksichtigen, entsprechend um Δx_{Level} vergrößert (Abb. 16.3). Im Anschluss wird mittels Punkt-In-Objekt-Methoden getestet, ob dieser innerhalb oder außerhalb des Geometrieobjektes liegt. Alle Knoten innerhalb des Festkörpers werden zu *solid* in der D3Q19BCMatrix des Blockdeskriptors gesetzt.

Für die anderen Knoten muss ermittelt werden, ob es sich eventuell um einen BC-Knoten handelt. Diese Knoten zeichnen sich dadurch aus, dass sie im Fluid unmittelbar am Rand der Geometrie liegen und in mindestens eine diskrete Richtung entweder keinen oder einen *solid* Nachbarn besitzen (Abb. 16.3). Die Bestimmung der BC-Knoten erfolgt durch die Berechnung des normierten Abstandes **q**, der separat für jede diskrete Richtung mit Hilfe von Punkt-Abstands-Methoden ermittelt wird. Die zeit- und speicherintensiven ClipLine-Methoden, die bei den Knotengittern zum Einsatz kamen, wurden im blockbasierten Code sukzessive durch performante, ressourcenschonende Raytracingalgorithmen ersetzt [4, 141].

Liegt q_i im Gültigkeitsbereich $]0,1]$, wird kontrolliert, ob sich an diesem Knoten bereits ein Randbedingungsobjekt in der D3Q19BCMatrix des Blockdeskriptors befindet, das ggf. aktualisiert wird. Ist keines vorhanden, so wird vom speziellen Deskriptor ein neues, modellabhängiges BC-Objekt erzeugt und zugewiesen. Dies gewährleistet die Verwendung unterschiedlicher von D3Q19BoundaryCondition abgeleiteter Randbedingungen für unterschiedliche Deskriptortypen. Ein Beispiel hierfür ist die MP3DBoundaryCondition, die zusätzlich über das Attribut wallPhi zur Bestimmung des Phasenfeldgradienten in Wandnähe verfügt. Der Interaktor selbst speichert neben

dem Block auch die Indizes der BC-Knoten, um u. a. die auf die Struktur ausgeübten Kräfte aus dem Fluid bestimmen zu können.

16.1.3 Zuweisung unterschiedlicher Randbedingungen

Um unterschiedliche Randbedingungen auf einfache Weise zur Verfügung zu stellen, verwenden D3Q19BmrInteractoren Randbedingungsbesucherklassen (Abb. 16.1). Diese setzen je nach Typ entweder richtungs- und/oder knotenspezifische Attribute im Randbedingungsobjekt des BC-Knoten. Für jede Richtung mit gültigem q_i wird die Methode adaptBCForDirection der jeweiligen Besucherklasse aufgerufen. Hier werden die richtungsspezifischen Attribute, wie z. B. BC-Flags oder q_i-Werte, gesetzt. Knotenglobale Werte, wie Druck oder der Phasenfeldwert wallPhi der Geometrie, können mit der Methode adaptBC gesetzt werden, die nur einmal pro BC-Knoten aufgerufen wird. Ein Interaktor kann mehrere Randbedingungsbesucherobjekte verwalten, wodurch verschiedene Besucherstrategien miteinander kombiniert werden können und wiederum Coderedundanz vermieden wird.

Derzeit verfügt VIRTUALFLUIDS über folgende für das Setzen von Randbedingungen relevante Besucherklassen:

- D3Q19NoSlipBCVisitor
  Dieser setzt die entsprechenden no-slip-Flags und **q** im Randbedingungsobjekt
- D3Q19SlipBCVisitor
  Neben den slip-Flags und dem **q** wird hier zusätzlich die für den BC-Algorithmus benötigte Knotennormale bestimmt (vgl. Abschn. 3.2).
- D3Q19DensityBCVisitor
  Für die Druckrandbedingung können konstante Dichten und/oder Druckfunktionen in Abhängigkeit von Ort und Zeit vorgegeben werden.
- D3Q19VelocityBCVisitor
  Analog zur Druckrandbedingung können für jede Richtung konstante Geschwindigkeiten und/oder Geschwindigkeitsfunktionen in Abhängigkeit von Ort und Zeit vorgegeben werden. Diese können wahlweise auch *periodisch* sein.
- MP3DPhiBCVisitor
  Diese Klasse setzt lediglich die im Mehrphasencode benötigte Phasenfelddichte ψ des Festkörperobjektes. Sie ist kombinierbar mit den anderen Besucherklassen.

Die Zeitabhängigkeit eines Interaktors wird über seine Randbedingungsbesucherobjekte definiert. In Quellcode 16.1 ist beispielhaft die Anwendung eines Interaktors mit zeitabhängigen Einflussprofil (inflowProfile) sowohl für den Ein- als auch für den Mehrphasenlöser dargestellt. Der Einfluss variiert die ersten 1.000 Zeitschritte und bleibt anschließend konstant. Somit wird ab Zeitschritt 1.001 die Zeitabhängigkeit des Besucherobjekts automatisch deaktiviert. Sind dann auch alle anderen Besucherobjekte des Interaktors zeitunabhängig, so werden für diesen die Aktualisierungsalgorithmen zur Laufzeit nicht mehr aufgerufen.

Die Funktionen innerhalb des D3Q19DensityBCVisitors bzw. des D3Q19VelocityBCVisitors werden mit dem Funktionsparser muParser von Ingo Berg [6] realisiert. Dieser ist so konstruiert, dass die Funktion nach der Initialisierung direkt in Bytecode konvertiert wird. Dadurch entfällt die zeitintensive Neuinterpretation zur Laufzeit. Variablen, wie die Knotenposition und die Zeit, werden dabei durch ihre Speicheradresse berücksichtigt.

Quellcode 16.1: *Anwendungsbeispiel eines Interaktors mit Geschwindigkeitsrandbedingung*

```
1  //Einfluss-Geometrie
2  GbCuboid3D inflowGeo( -5, 0, 0/*Ecke(W,S,B)*/, 0, 10, 10/*Ecke(E,N,T)*/ );
3
4  //Einflussprofil (Anm: x1, x2, x3 sowie t sind geschützte Variablen)
5  mu::Parser inflowProfile;;
6  inflowProfile.SetExpr( "16*vx1*x2*x3*(H-x2)*(H-x3)/(H^4)*t/1000");
7  inflowProfile.DefineConst( "vx1", 0.05);
8  inflowProfile.DefineConst( "H", inflowGeo.getHeight() );
9
10 //Visitorobjekt erstellen
11 D3Q19VelocityBCVisitor velVisitor( inflowProfile, 0/*startTime*/, 1000/*endTime*/ )
12
13 //Einphase
14 D3Q19BMRInteractor epInteractor( inflowGeo, epBlockGrid, velVisitor );
15 epBlockGrid.addAndInitInteractor( epInteractor );
16
17 //Mehrphase
18 D3Q19BMRInteractor mpInteractor( inflowGeo, mpBlockGrid, velVisitor );
19 interactor.addBCVisitor( MP3DPhiBCVisitor(-1.0/*wallPhi*/) );
20 mpBlockGrid.addAndInitInteractor( mpInteractor );
```

16.2 Dreiecksnetzinteraktor

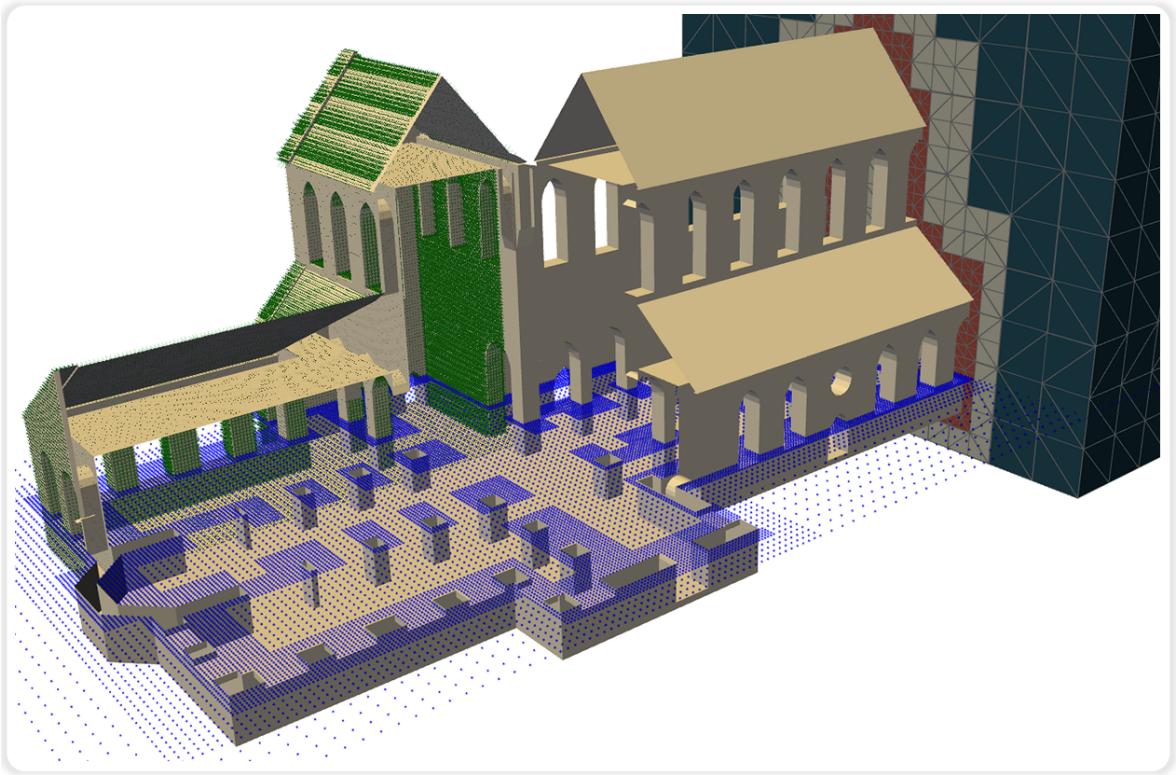


Abbildung 16.4: *Blockgitterverfeinerung (virtuelle Rekonstruktion des Klosters Walkenried)*

Der D3Q19TriFaceMeshInteractor wurde für die Verwendung von Dreiecksnetzen optimiert, die vor allem bei komplexen Geometrien, wie Kühltürmen, Kraftfahrzeugen und Kirchen, zur Anwendung

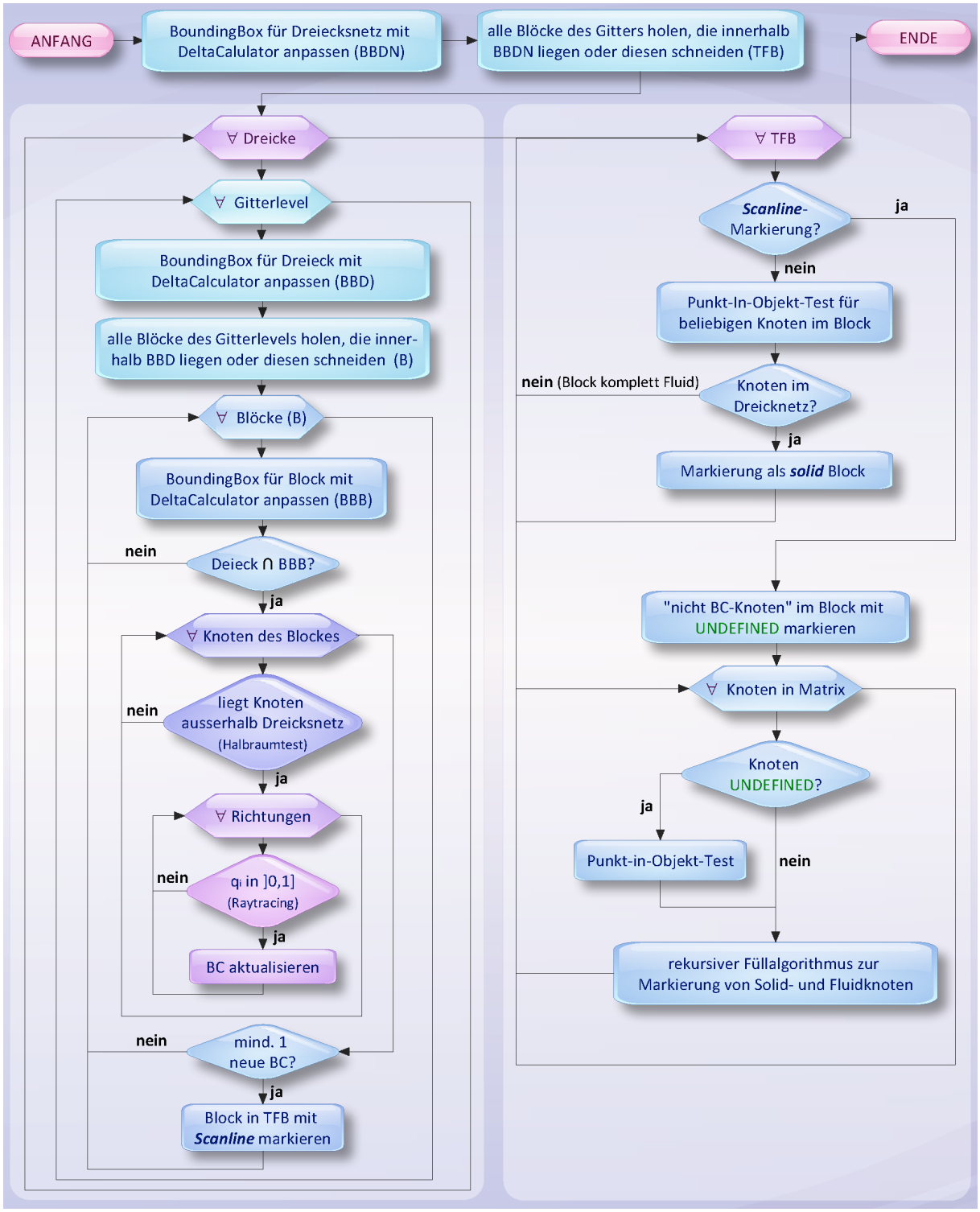


Abbildung 16.5: *Flussdiagramm zur Ermittlung der BC-Knoten*

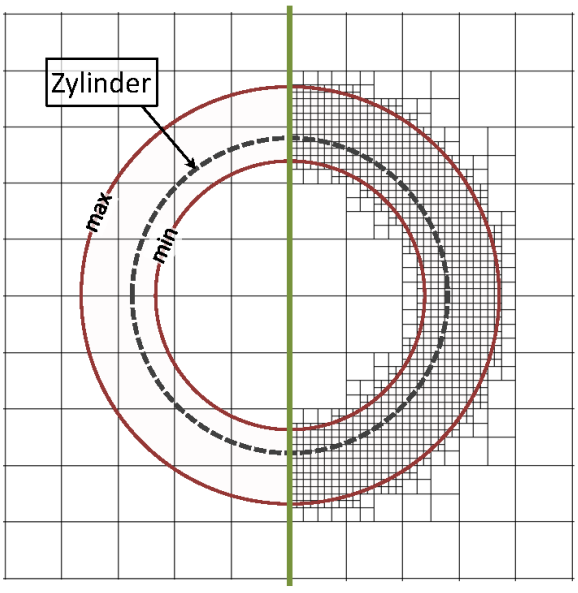
kommen (Abb. 16.4). Der Standardinteraktor weist bei solchen Netzen aufgrund des zeitaufwändigen Punkt-in-Objekt-Tests für Dreiecksgitter selbst unter Verwendung effizienter Implementierungen [113] eine vergleichsweise schlechte Performance auf. Bei statischen Strukturen kann eine octreebasierte Speicherung des Netzes innerhalb der entsprechenden Geometrieklasse, die durch entsprechenden Präprozess eine schnellere Klassifizierung der Knoten erlaubt, Abhilfe schaffen.

Der Dreiecksnetzinteraktor verwendet eine andere Vorgehensweise. Hier werden zunächst mittels Raytracing- und Halbebenentests die BC-Knoten der Dreiecke bestimmt und anschließend die inneren Knoten der Struktur mittels rekursiver Füllalgorithmen identifiziert. Der vollständige Initialisierungsalgorithmus des D3Q19TriFaceMeshInteractors ist in Form eines Flussdiagrammes in Abb. 16.5 dargestellt.

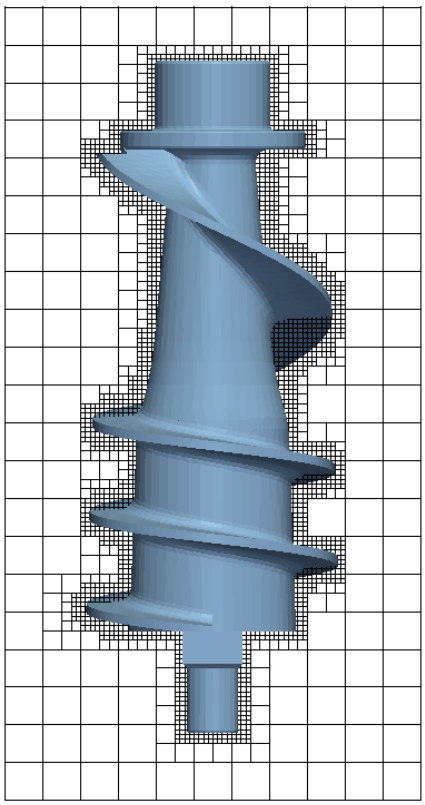
16.3 Sonstige Funktionalitäten

16.3.1 Gittergenerierung

Wahlweise kann mit Hilfe der Interaktoren das Gitter im Bereich der Festkörperoberfläche verfeinert werden. Der Anwender kann dabei den minimalen und maximalen Abstand zur Oberfläche angeben, der verfeinert werden soll. Für geometrische Primitive werden hierzu zwei Klone des Geometrieobjektes erstellt, die anschließend entsprechend skaliert werden. Alle Blöcke, die sich innerhalb des äußeren Klons befinden bzw. diesen schneiden und nicht vollständig innerhalb der inneren geklonten Geometrie liegen, werden im Anschluss verfeinert (Abb. 16.6a).



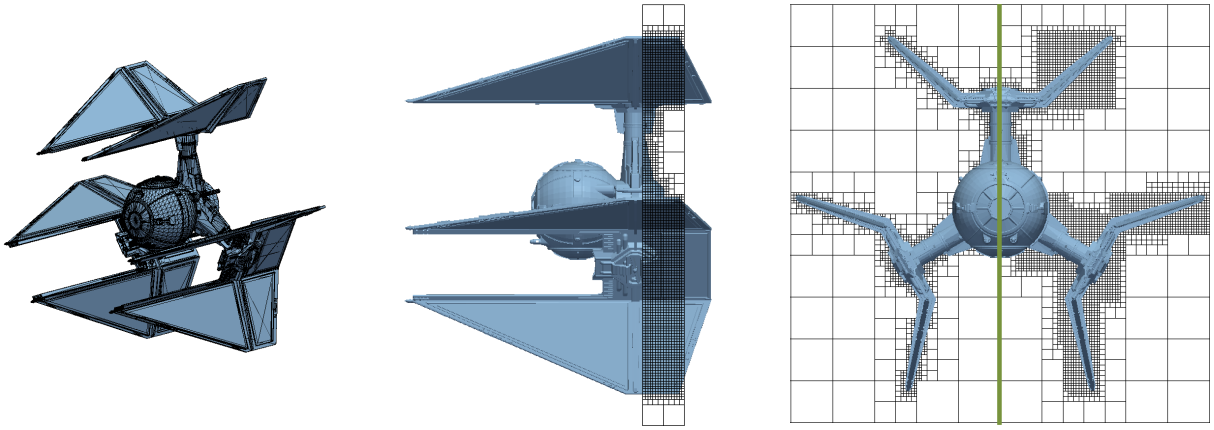
(a) Zylinder mit skalierten Klongeometrien



(b) Triangulierte Mauerkrone aus [1]

Abbildung 16.6: *Beispiele für Blockverfeinerungen mit dem Dreiecksnetzinteraktor (Schnitt)*

Im Gegensatz zur Blockbesucherklasse Bmr3DRefineBetweenTwoGeos, die ebenfalls den Bereich zwischen zwei geometrischen Objekten verfeinert, iteriert der Interaktor nicht über alle Blöcke, sondern untersucht nur die notwendigen. Der Standardinteraktor überprüft hierfür alle Blöcke der BoundingBox einer Geometrie. Beim D3Q19TriFaceMeshInteractor werden die zu verfeinernden Blöcke mittels BoundingBox-, Halbraum- und Dreiecksüberlappungsalgorithmen bestimmt.



(a) *Dreicksnetz* ***(b)*** *Blockgitter (Schnitt)* ***(c)*** *Blockgitter erweitert/standard*

Abbildung 16.7: *Blockverfeinerung mit D3Q19TriFaceMeshInteractor für einen TIE Defender aus [2]*

In Abb. 16.7c ist exemplarisch der Unterschied zwischen einer reinen BoundingBoxverfeinerung der einzelnen Dreiecke und der erweiterten Verfeinerung aufgezeigt, die insbesondere bei großen, nicht achsenorthogonalen Dreiecken zur Geltung kommt. Die Effizienz der Gittergenerierung ist abhängig vom Verhältnis der BoundingBox eines Dreiecks zur Größe eines Blockes. Die Diskretisierungsdauer für die dargestellten Beispiele (≈ 100.000 Dreiecke) lag deutlich unter einer Sekunde auf einer *Intel® Core™ 2 Quad Q6600 CPU @2,4GHz*, wobei nur ein Kern in Anspruch genommen wurde. Um gültige Berechnungsgitter zu erhalten, müssen diese Gitter anschließend mit Hilfe entsprechender Besucherklassen geglättet werden.

16.3.2 Kraftauswertung

Bei der Initialisierung bzw. Aktualisierung eines Interaktors werden die Blöcke und Knotenindizes der zur Geometrie gehörenden Randknoten gespeichert. Die Kraft wird in den Standardinteraktoren über die Impulsaustauschmethode ermittelt (vgl. Kap. 7). Spezielle Interaktoren, wie der D3Q19FemInteractor, verwenden hierfür die aufwändigere Spannungsintegrationsmethode [46].

16.3.3 Fluid-Struktur-Interaktion (SteeringPlugin)

Mit Hilfe der optionalen anwendbaren SteeringPlugins ist es möglich Geometrieobjekte zur Laufzeit zu verändern (z. B. durch Verschieben oder Verformen). Hierfür definiert der Anwender das Verhalten des jeweiligen Objektes durch die Implementierung der virtuellen Methode moveGbObject der entsprechenden SteeringPlugin-Klasse. Diese wird zur Laufzeit vor der Aktualisierung der Interaktoren durchgeführt. So kann zum Beispiel eine Starrkörperverschiebung entlang eines festen Pfades realisiert werden. Über diese Schnittstelle wird auch die Fluid-Struktur-Interaktion realisiert. Hierbei werden zunächst die Kräfte aus dem Fluid an einen Strukturlöser übergeben, der anschließend

die resultierende Verschiebung berechnet. Diese Funktionalität ist bis einschließlich der Veränderung der Geometrie im SteeringPlugin abgekapselt [46]. Die Aktualisierung des Strömungsfeldes und der Randbedingungen wird anschließend vom Interaktor durchgeführt. Auf die SteeringPlugins wird später bei der Parallelisierung genauer eingegangen (vgl. Abs. 17.7.4).

16.3.4 Unterstützung von OpenMP

Sämtliche Initialisierungsalgorithmen der Interaktorklassen wurden mittels OpenMP parallelisiert, um die zusätzlichen Rechenressourcen moderner Mehrkern-CPUs nutzen zu können. Für die in diesem Kapiteln verwendeten Dreiecksnetze wurde auf einer *Intel® Core™ 2 Quad Q6600 CPU @2,4GHz* unter Verwendung aller vier Kerne im Schnitt die 3,4-fache Leistung gegenüber einem Kern erreicht. Ein leistungsbeschränkender Grund ist hierbei die Verwendung der nicht threadsicheren STL-Bibliothek, wodurch nur blockierende Zugriffe auf die STL-Container möglich sind.

17 Verteiltes, servicebasiertes Framework

Das verteilte Framework muss eine Reihe von Anforderungen zu erfüllen. Hierzu gehören u. a. eine Kopplungsschnittstelle für Strukturlöser, eine Interaktionsschnittstelle für externe Anfragen, die Auswertung von Strömungsdaten sowie eine dynamische adaptive Gittersteuerung. Aus diesem Grund fiel die Entscheidung bei dem hier entwickelten parallelen Löser auf eine serviceorientierte Architektur (SOA). Diese kapselt unterschiedliche Funktionalitäten, wie die Verwaltung von Geometrieobjekten oder die Strömungsberechnung auf einem Teilgitter, in verschiedenen Services bzw. Diensten. Diese sind in sich abgeschlossen, können eigenständig lokal oder über das Netzwerk genutzt werden und stellen zur Interaktion öffentliche Schnittstellen zur Verfügung [85]. Die einzige Einschränkung gegenüber der klassischen SOA-Definition ist die fehlende, nicht benötigte Sprachunabhängigkeit. Wie in Kap. 13 beschrieben, stellt RCF hierfür die geforderte Funktionalität zur Verfügung.

17.1 Übersicht über die Services

Eine Übersicht über die Komponenten des in dieser Arbeit entwickelten Serviceframeworks gibt Abb. 17.1. Hierbei ist zu beachten, dass die Services untereinander ausschließlich über das RCF kommunizieren. Intern arbeiten sie multithreaded, um parallel Anfragen bearbeiten zu können. Kritische Abschnitte, in denen gemeinsam genutzte Datenstrukturen bearbeitet werden, sind mit Hilfe wechselseitiger Ausschlüsse (mutex) geschützt. Da jeder Service sowohl Client als auch Server ist, entspricht dies einem Peer-to-Peer Netzwerk, in dem alle Beteiligten gleichberechtigt sind.

Um einen effizienten numerischen Datenaustausch zu gewährleisten, wurde dieser in der Berechnungsklasse von der RCF-Implementierung entkoppelt und frei über das Connector-Transmitter-Konzept wählbar gemacht. Dadurch kann dieser mit einer beliebigen Bibliothek erfolgen, insofern hierfür die entsprechenden Transmitterklassen zur Verfügung stehen.

Folgende Dienste kommen in VIRTUALFLUIDS zum Einsatz:

- **IPService**
  Alle Services registrieren sich beim Start bei diesem rudimentären Namensdienst und übermitteln ihren Servicenamen sowie IP-Adresse und Port, über die sie zu erreichen sind. Dies ist notwendig, um die Verbindungsdaten der einzelnen Dienste in einem verteilten System global und dynamisch zur Verfügung stellen zu können. Es kann vorkommen, dass die Verbindungsdaten des IPService beim Start der anderen Dienste unbekannt sind. Dies ist insbesondere bei Einsatz auf Großrechnern der Fall. Dort werden Programme oftmals mit Hilfe von Queueing-Systemen gestartet, auf die der Anwender meist nur eingeschränkt Einfluss hat (z. B. HLRB II - SGI Altix 4700 des LRZ in München). Hier kann der IPService wahlweise seine eigenen Verbindungsdaten kontinuierlich über eine Multicast- oder Broadcastadresse bekannt geben, die dann von den anderen Diensten entsprechend verarbeitet werden.

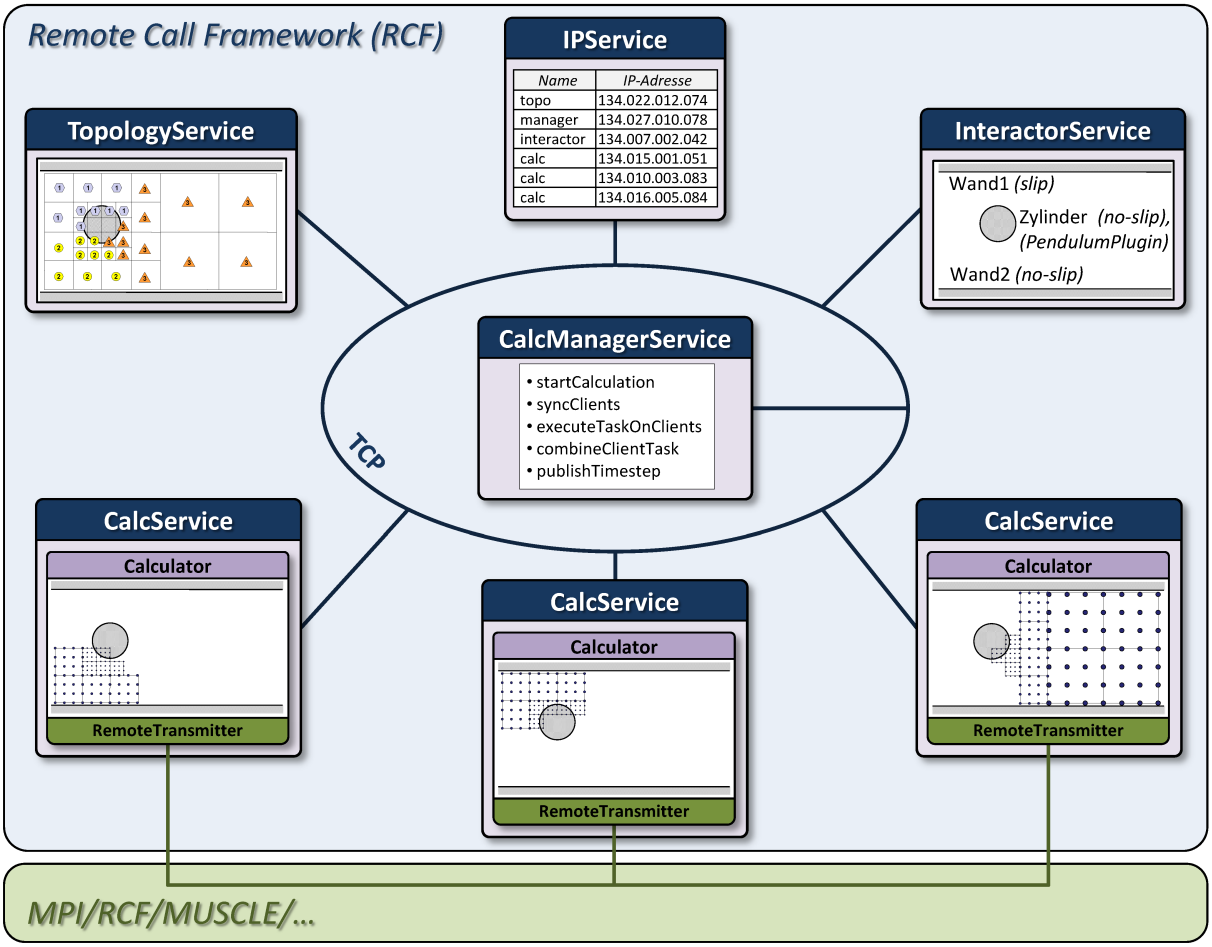


Abbildung 17.1: *Services von* VIRTUALFLUIDS

- **InteractorService**
  Dieser erweiterte Geometrieservice stellt die verwendeten Interaktoren inklusive Geometrieobjekt, Geometrietyp, Randbedingungs- und SteeringPlugin-Modulen zur Verfügung (vgl. Kap. 16). Zur Berechnungslaufzeit erfolgen Objektverschiebungen über diesen zentralen Service. Die anderen Dienste bearbeiten ausschließlich lokale Kopien der Interaktoren, die z. B. bei Simulationen mit einer Fluid-Struktur-Interaktion die resultierende Teilkräfte auf die jeweiligen Geometrieobjekte an den InteractorService senden, der diese dann aufsummiert und anschließend mit den entsprechenden SteeringPlugin-Modulen die resultierenden Verschiebungen berechnet, die er dann an die Kopien auf den Rechenservices zurücksendet.

- **TopologyService**
  Hier wird die gesamte Topologie des Rechengitters einer Simulation vorgehalten. Sämtliche topologieverändernden Operationen werden von diesem Service durchgeführt: Gitterverfeinerung, Ermittlung der aktiven und nicht-aktiven Blöcke, Gebietszerlegung, die Übertragung der einzelnen Segmente und der notwendigen Informationen für den Datenaustausch zu den entsprechenden CalcServices, adaptive Gitterverfeinerung/-gröberung etc.

  Optional wird entweder ein reines Blockgitter ohne Blockdeskriptoren oder ein hybrides, mit Knoteninformationen versehenes Blockgitter verwendet. Letzteres wird benötigt, um Dreiecksnetze konsistent diskretisieren zu können und die Blockwichtungen für die Partitionierung korrekt be-

stimmen zu können. Um den Speicherbedarf zu minimieren werden hier im Unterschied zum Berechnungsgitter ausschließlich Blockdeskriptoren verwendet, die außer einer optimierten Randbedingungsmatrix keinerlei physikalische Daten beinhalten.

- **CalcService**
  In diesem erfolgt die eigentliche Strömungsberechnung mit der modellabhängigen Calculatorklasse. Steht nur ein CalcService zur Verfügung, dann entspricht dies einer seriellen Simulation. Dem Dienst wird durch den TopologyService das entsprechende (Teil-)Blockgitter zugewiesen. Nach Initialisierung der modellspezifischen Blockdeskriptoren, der lokalen und verteilten Kommunikationsmodule, des Calculators und der Zuweisung der physikalischen Gittereigenschaften, wird die numerische Simulation durchgeführt. Der Austausch der numerischen Daten kann hier, wie in Kap. 15 beschrieben, vom RCF entkoppelt mit alternativen Interprozessbibliotheken, wie z. B. MPI oder MUSCLE [71, 72, 78], erfolgen.

- **CalcManagerService**
  Über diese zentrale Steuereinheit können andere Dienste Aufgaben (z. B. in Form von Block- oder Gitterbesucherobjekten) an die Rechendiensten übertragen. Dies kann auch zur Rechenlaufzeit erfolgen, wobei verschiedene Synchronisierungspunkte innerhalb der Zeitschleife angegeben werden können, zu denen die jeweilige Operation durchgeführt werden soll. Diese Schnittstelle ermöglicht beispielsweise Dritten (z. B. externen Viewern) die Ermittlung der aktuellen Maximalgeschwindigkeit oder anderer Strömungseigenschaften. Je nach Aufgabentyp muss der aufrufende Service die Antwort der Clients abwarten (Zweiwegeaufruf) oder nicht (Einwegaufruf).

  Der Manager stellt zudem einen Publisherdienst zur Verfügung, der z. B. allen registrierten Abonnenten (Subscriber) fortlaufend den aktuellen Berechnungszeitschritt übermittelt. Über den CalcManagerService wird die Berechnung gestartet und kann von den Clients optional zur Synchronisierung verwendet werden. Auch seitens der Rechendienste kann der Manager genutzt werden, um verteilte Aufgaben zentral zu kombinieren und das Ergebnis wieder zu verteilen (z. B. das Erstellen einer Postprozessmetadatei, bei der die Klienten die einzutragenden lokalen Dateinamen übermitteln).

17.2 Start der Services

Zu Beginn einer Simulation müssen die benötigten Dienste auf dem Rechner initialisiert werden, auf dem später Anfragen entgegengenommen und bearbeitet werden sollen. Dies kann beispielsweise über einen Masterprozess (z. B. in Form einer GUI) erfolgen. Bei der softwaretechnischen Umsetzung wurde hinsichtlich der Benutzerfreundlichkeit darauf geachtet, dass die zu verwendende Syntax ähnlich zu der in seriellen Anwendungen ist, die auch weiterhin ohne Verwendung des Serviceframeworks durchgeführt werden können.

In Abb. 17.2 ist das Sequenzdiagramm für die Inbetriebnahme diverser Services dargestellt. Zunächst wird mit dem IPService der Namensdienst gestartet. In diesem Beispiel gibt dieser anschließend kontinuierlich seine Verbindungsdaten über Multicastnachrichten bekannt. Mit Hilfe dieser verbindungslosen Nachrichten können sich anschließend die übrigen Services beim IPService registrieren. Sobald die benötigten Dienste gestartet wurden, kann der Multicast-Thread beendet werden und der Anwender mit dem Simulationssetup beginnen.

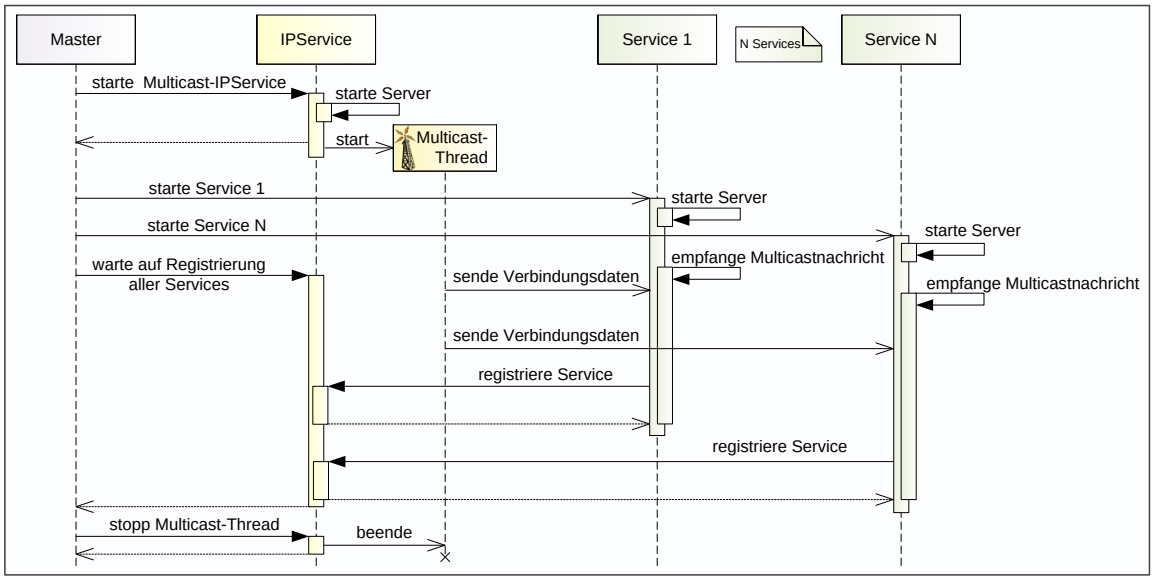


Abbildung 17.2: *Start der Services mit Multicast beim* IPService *(UML-Sequenzdiagramm)*

17.3 Initialisierung der Topologie

Bei der Initialisierung der Topologie muss sowohl das Blockgitter als auch die mittels Interaktoren realisierten Randbedingungen konfiguriert werden. Die Blockgenerierung in der parallelen Programmversion erfolgt dabei genau wie in der seriellen.

Quellcode 17.1: *RefineVisitor*

```
1  //Verbinden mit dem TopologyService (Proxy erstellen):
2  RCF::TcpEndpoint connectionInfo("134.169.81.214" /*IP*/, 5000/*Port*/);
3  RcfClient< ITopologyService > topoProxy( connectionInfo );
4
5  //Erzeugen eines Gitters mit 10x4x4 Bloecken und 100x40x40 Knoten pro Block:
6  topoProxy.constructBlockGrid( 10,4,4,  100,40,40,  0/*Basislevel*/ );
7
8  //Gebietsverfeinerung:
9  GbSphere3D sphere( 10/*x1*/, 10/*x2*/, 10/*x3*/, 5/*rad*/ );
10 RefineVisitor refineVisitor( sphere, 0/*startLevel*/, 3/*stopLevel*/);
11 topoProxy.adaptByVisitor( refineVisitor );
12
13 //Abfrage des feinsten Levels:
14 int finestLevel = topoProxy.getFinestLevel();  // -> 4
```

Quellcode 17.1 veranschaulicht exemplarisch anhand einer Blockgitterinitialisierung auf dem TopologyService, wie man sich mit einem Service verbindet und diesen anschließend verwendet. Zuerst erzeugt man mit Hilfe der in *connectionInfo* gespeicherten Verbindungsdaten, die beispielsweise vom IPService bezogen werden, das Proxyobjekt *topoProxy*, über das später auf den gewünschten Dienst zugegriffen wird *(Zeile 2-3)*. Bei dessen Erstellung wird eine TCP/IP-Verbindung zu dem eigentlichen TopologyService aufgebaut. Sollte diese fehlschlagen, wird eine entsprechende Fehlermeldung generiert.

Nach erfolgreicher Verbindung können auf dem Proxyobjekt alle in der Interfaceklasse *ITopologyService* des TopologyService deklarierten Methoden aufgerufen werden. In dem Beispiel wird zunächst

ein Blockgitter erzeugt *(Zeile 6)*. Hierzu werden die benötigte Anzahl an Blöcken je Achsenrichtung, die Dimension der Knotenmatrix pro Block und der Basislevel übergeben. Für ein gültiges Berechnungsgitter muss hierbei beachtet werden, dass die Seitenverhältnisse eines Blockes mit denen der Knotenmatrix übereinstimmt. Dadurch wird gewährleistet, dass die Knotenabstände Δx_1, Δx_2 und Δx_3 identisch sind. Die Hexaeder repräsentierenden Blöcke müssen dabei keine Würfelform aufweisen.

Zur Anpassung des Gitters stellt die Interfaceklasse des TopologyService die wesentlichen Methoden des Bmr3DBlockGrids zur Verfügung. Die wichtigste hierbei ist adaptByVisitor, mit deren Hilfe der Anwender beliebige Block- und Gitterbesucherklassen des seriellen Lösers verwenden kann. In dem Beispiel wird das Gitter mit Hilfe des *refineVisitor*-Moduls im Bereich der Kugel *sphere* verfeinert *(Zeile 9-11)*.

Im Gegensatz zu dem gleichartigen Beispiel in Abschn. 6.3 werden hier die Methoden nicht im aufrufenden Prozess, sondern auf dem (entfernten) Service ausgeführt (Abb. 17.2). Damit dort die Methoden durchgeführt werden können, müssen die entsprechenden, serialisierbaren Argumente, wie z. B. das Besucherobjekt und die Kugel, zum TopologyService gesendet werden. Nach erfolgreicher Bearbeitung wird der entsprechende Rückgabewert zurück zum Proxy gesendet. In dem Beispiel ist dies der Wert des feinsten Gitterlevels *(Zeile 14)*.

An dem Beispiel lässt sich gut erkennen, dass die Terminologie des RCF dem Standard-C++ entspricht und somit die Verwendung des parallelen Frameworks nur wenig Einarbeitungszeit benötigt. Auch lassen sich selbst komplexe Klasseninstanzen unter Vorhandensein entsprechender Serialisierungsmethoden sehr einfach zwischen zwei Prozessen austauschen (vgl. Abs. A5.3).

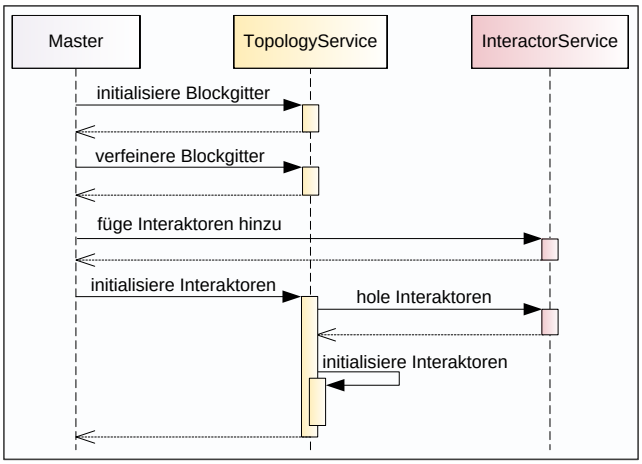


Abbildung 17.3: *Initialisierung der Topologie (UML-Sequenzdiagramm)*

Hinsichtlich der Integration von Randbedingungen unterscheidet sich die Vorgehensweise beim verteilten Löser vom seriellen. Hier können die Geometrien sowie deren Randbedingungen nicht direkt dem Gitter hinzugefügt werden. Zur Trennung der Verantwortlichkeiten übernimmt deren Verwaltung der zentrale InteractorService, der auch bei der Interaktion mit Strukturlösern zur Anwendung kommt. Alle anderen Services fordern bei Bedarf Kopien der Interaktoren an und arbeiten mit diesen.

Der Anwender erstellt wie gewohnt die gewünschte Geometrie und fügt sie zusammen mit den Randbedingungsobjekten einem Interaktor hinzu. Dessen Instanz wird anschließend nicht direkt dem Blockgitter des TopologyServices, sondern dem InteractorService zugewiesen. Sobald alle benötigten Interaktoren erstellt und übertragen wurden, kann sich der TopologyService diese besorgen und in das Gitter integrieren (Abb. 17.3).

17.4 Gebietszerlegung

Bevor die verteilte Berechnung gestartet werden kann, muss eine Gebietszerlegung durchgeführt werden. Wie bereits im Knotencode wurde die Segmentierung beim Blockgitter durch die Verwendung von Gitterbesucherklassen modular gehalten. Somit kann der derzeit hauptsächlich verwendete, auf der METIS-Bibliothek basierende DCMetisGridVisitor durch andere Klassen mit alternativen Verfahren ersetzt werden.

Das an den TopologyService gesendete DCMetisGridVisitor-Modul erstellt dort den für die Segmentierung erforderlichen METIS-Graphen (vgl. Abschn. 10.3). Jeder Block des Gitters entspricht dabei einem Knoten und jede Interblockkommunikationsrichtung einer Kante des Graphen. Durch die vergleichsweise geringe Anzahl an Blöcken gegenüber der tatsächlichen Knotenanzahl ist der Nachteil des hohen Speicherbedarfs für den METIS-Graphen (vgl. Abschn. 10.6) und der nachfolgenden Zerlegung nicht mehr gegeben. Die Besucherklasse wurde so allgemein gehalten, dass die für den Graphen zu berücksichtigenden Knotenwichtungen sowie die Kantenrichtungen und deren Wichtungen frei definiert werden können. Für Simulationen mit sich verändernden Strukturkörpern können bei der Segmentierung optional auch nicht-aktive Blöcke berücksichtigt werden, da diese u. U. zur Laufzeit aktiviert werden.

Für die Physikschicht ergeben sich die Knotenwichtungen aus der Anzahl der im Block enthaltenen Fluidknoten sowie der internen Iterationen je gröbstem Zeitschritt. Die notwendigen Kanten hängen vom jeweiligen Modell ab. Während das 3-D-Standardeinphasenmodell ausschließlich über die diskreten $d3q19$-Richtungen $\mathbf{e}_i$ kommuniziert, werden im Mehrphasenkern aufgrund von Ghostlayern zusätzlich Daten in Richtung der Raumdiagonalen ausgetauscht. Die Kantenwichtung resultiert aus der Anzahl der Knoten, die sich auf der jeweiligen Grenzschicht befinden (vgl. Tab. 17.1). Im Gitterübergangsbereich werden diese Wichtungen unter Beachtung des erhöhten Kommunikationsaufwandes um den Faktor zwei erhöht.

Kommunikationsrichtung	Einphase	Einphase*	Mehrphase
über $x_1 - x_2$-Blockfläche	1.000	100	1.000
über $x_1 - x_3$-Blockfläche	500	50	500
über $x_2 - x_3$-Blockfläche	200	20	200
über x_1-Blockkante	50	5	50
über x_2-Blockkante	20	2	20
über x_3-Blockkante	10	1	10
über Blockecke	-	-	1

* um den kleinsten gemeinsamen Teiler reduzierte Wichtung

Tabelle 17.1: *Kantenwichtungen eines Blocks mit 50x20x10 Knoten*

Die METIS-Bibliothek selbst wird in VIRTUALFLUIDS durch das *Fassaden*-Entwurfsmuster gekapselt [38]. Dadurch steht dem Programmierer eine MetisGraph-Klasse und verschiedene Partitionierungsmethoden zur Verfügung. In Abhängigkeit von der Zahl der zu erstellenden Partitionen n wird gemäß der Empfehlung des Entwicklers entweder die METIS_PartGraphRecursive- ($n < 9$) oder die METIS_PartGraphKway-Funktion ($n > 9$) mit den jeweiligen Standardoptionen verwendet [89]. Beide haben eine gleichmäßige Gesamtwichtung pro Segment unter Minimierung der zerschnittenen Kanten zum Ziel. Für heterogene Systeme existieren äquivalente Funktionen, bei denen die Leistungsunterschiede der einzelnen Komponenten mit Hilfe von Gewichtungsfaktoren berücksichtigt werden können. Nach der Zerlegung des Graphen werden die zugewiesenen Knotensegmentnummern in dem jeweiligen Block gespeichert.

17.5 Initialisierung der Berechnungsgitter

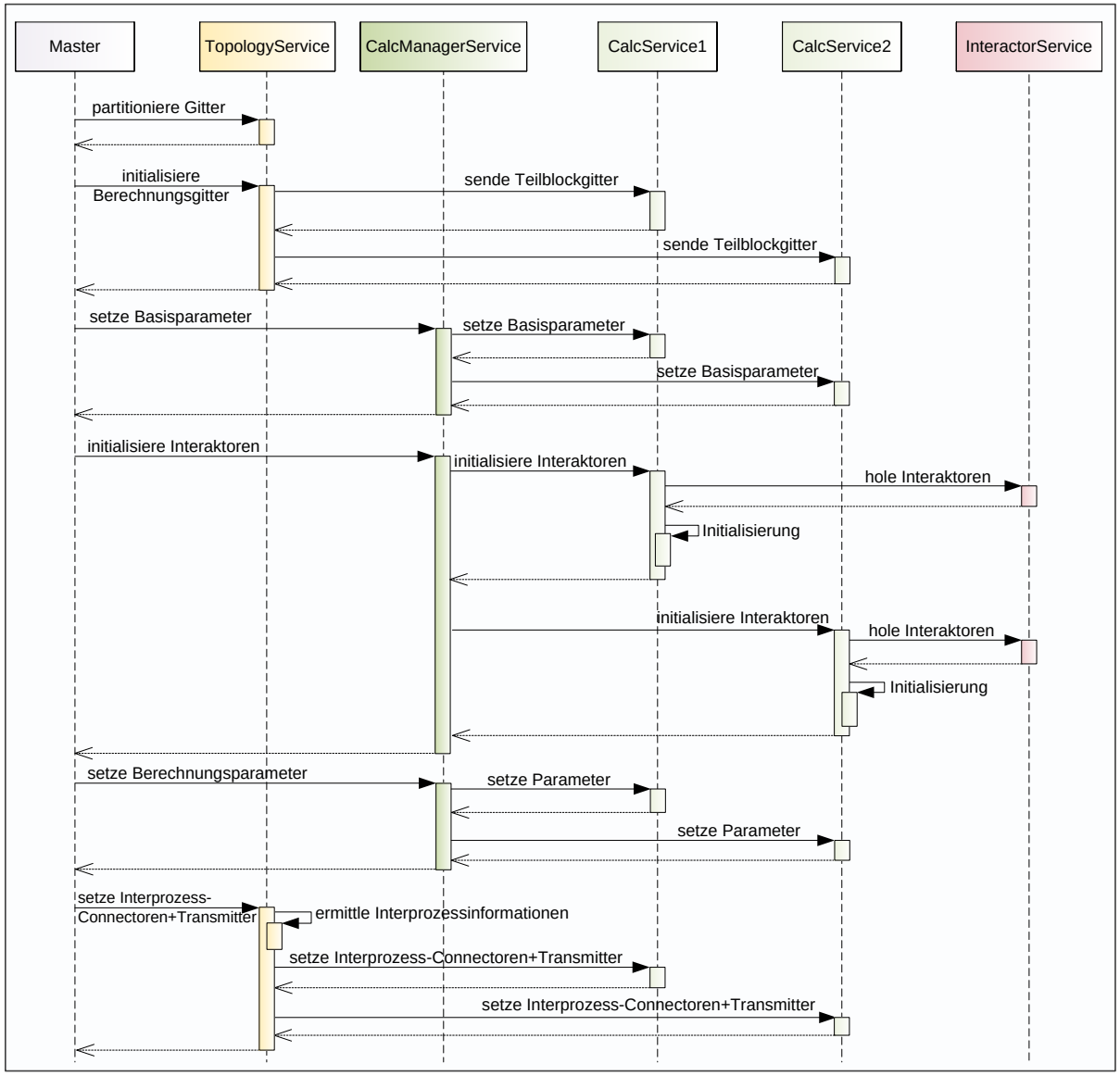


Abbildung 17.4: *Initialisierung der Berechnungsgitter (UML-Sequenzdiagramm)*

Nach der Gebietszerlegung ermittelt der TopologyService die zusammengehörenden Blöcke eines Segments und sendet diese zu den einzelnen CalcServices (Abb. 17.4). Dort werden das spezifische Gitter erstellt, die Blöcke einsortiert und die modellspezifischen Blockdeskriptoren hinzugefügt. Das jeweilige Gitter wird dabei mit Hilfe von Modulen erzeugt, die gemäß dem Erzeugungsmuster *Abstrakte Fabrik* (*engl.:* abstract factory) implementiert wurden [43], da die CalcServices grundsätzlich unabhängig vom verwendeten Physikmodell sind. Der Anwender muss deshalb beim Start des Dienstes das zum verwendeten Modell korrespondierende Erzeugermodul des benötigten Blockgitters (Einphasen-, Mehrphasengitter, etc.) übergeben.

Die restlichen für die Initialisierung des Rechengitters nötigen Operationen erfolgen über den CalcManagerService, der als zentrale Steuereinheit fungiert und dedizierte Anfragen und Aufga-

ben an die CalcServices weiterleitet. Wie in Abb. 17.4 veranschaulicht, werden zunächst einige Basisparameter, wie das zu verwendende Kollisionsmodell, an den CalcManagerService übergeben. Von diesem aus werden sie dann an die einzelnen Klienten übermittelt und dort zugewiesen. Im Anschluss daran werden die Interaktoren in die jeweiligen Teilgitter eingefügt. Die Anweisung hierfür erfolgt wiederum über den CalcManagerService. Die Rechendienste verbinden sich daraufhin mit dem InteractorService und beziehen die einzelnen Interaktoren. Bei deren Initialisierung auf den Rechendiensten werden nun zusätzlich die Randbedingungsobjekte der Berechnungsknoten in den Blöcken gesetzt.

Vor dem letzten Schritt werden mit Hilfe diverser Besucherklassen, die größtenteils auch im seriellen Code verwendet werden, unter anderem die Knotenverteilungen initialisiert, die lokalen Connectoren und Transmitter gesetzt, zeitabhängige Gittermodule, wie z. B. eine zeitgesteuerte Veränderung der Viskosität, hinzugefügt oder die zu verwendenden Schreibmodule für den Postprozess definiert.

Nachdem die lokalen Connectoren gesetzt wurden, werden nun die restlichen Connectoren und sämtliche Interprozesstransmitter gesetzt. Hierzu ermittelt der TopologyService für jede Datenaustauschrichtung die Blocknachbartupel, die unterschiedliche Segmentnummern aufweisen. Diese segmentweise gesammelten Daten werden anschließend zu den CalcServices geschickt, die mit Hilfe dieser Informationen die entsprechenden Transmittermodule erstellen und den Connectoren zuweisen.

17.6 Ablauf der Berechnung

Nachdem alle für den Rechenprozess notwendigen Daten auf den Klienten verfügbar sind, wird die Simulation vom Masterprozess aus mit Hilfe des CalcManagerServices initiiert (Abb. 17.5). Zuerst werden die später zur Kommunikation benötigten Proxys der anderen Services erstellt und die interne Datenstruktur aufgebaut. Im Anschluss werden die Connectoren initialisiert. Dabei werden u. a. die Puffer der Transmittermodule eingerichtet, wobei der Datenaustausch erstmals über die Transmitter und somit nicht über das Serviceframework erfolgt. MPI-Transmitter gleichen in diesem Schritt ihre Sende- und Empfangsfelder ab.

Bevor die eigentliche Zeitschleife beginnt, synchronisieren sich die Klienten über den CalcManagerService, indem sie auf dessen Proxy die sync Methode aufrufen. Dieser prozessblockierende Aufruf ist erst dann beendet, nachdem der CalcManagerServices alle RPCs der Rechendienste empfangen hat.

In der gestaffelten Berechnungsschleife werden dann die Lattice-Boltzmann-Routinen, wie Kollision, Propagation und Skalierung, durchgeführt (Alg. 3). Die Propagation über die Blockgrenzen erfolgt dabei durch den numerischen Datenaustausch der Connectoren zum Teil unter Einsatz entsprechender Interprozesskommunikation. Hierbei wurde darauf geachtet, dass die Transmitter selbstsynchronisierend arbeiten, um zusätzliche zeitintensive globale Synchronisierungspunkte, wie z. B. durch die Verwendung einer *Barrier* bei MPI, zu vermeiden. Hierzu senden die Transmitter die Daten nichtblockierend und empfangen diese blockierend oder umgekehrt. Ein gekoppelter *SendReceive*-Aufruf, bei dem zwei Prozesse im selben Schritt sowohl Daten senden als auch empfangen, ist in der derzeitigen Implementierung nicht vorgesehen. Zum einen müsste hier zur Vermeidung von Deadlocks im Präprozess die Reihenfolge des Austausches für alle Prozesse definiert werden, und zum anderen können bei einem asynchronen Datenaustausch mögliche Leerlaufzeiten für die Durchführung anderer Algorithmen genutzt werden.

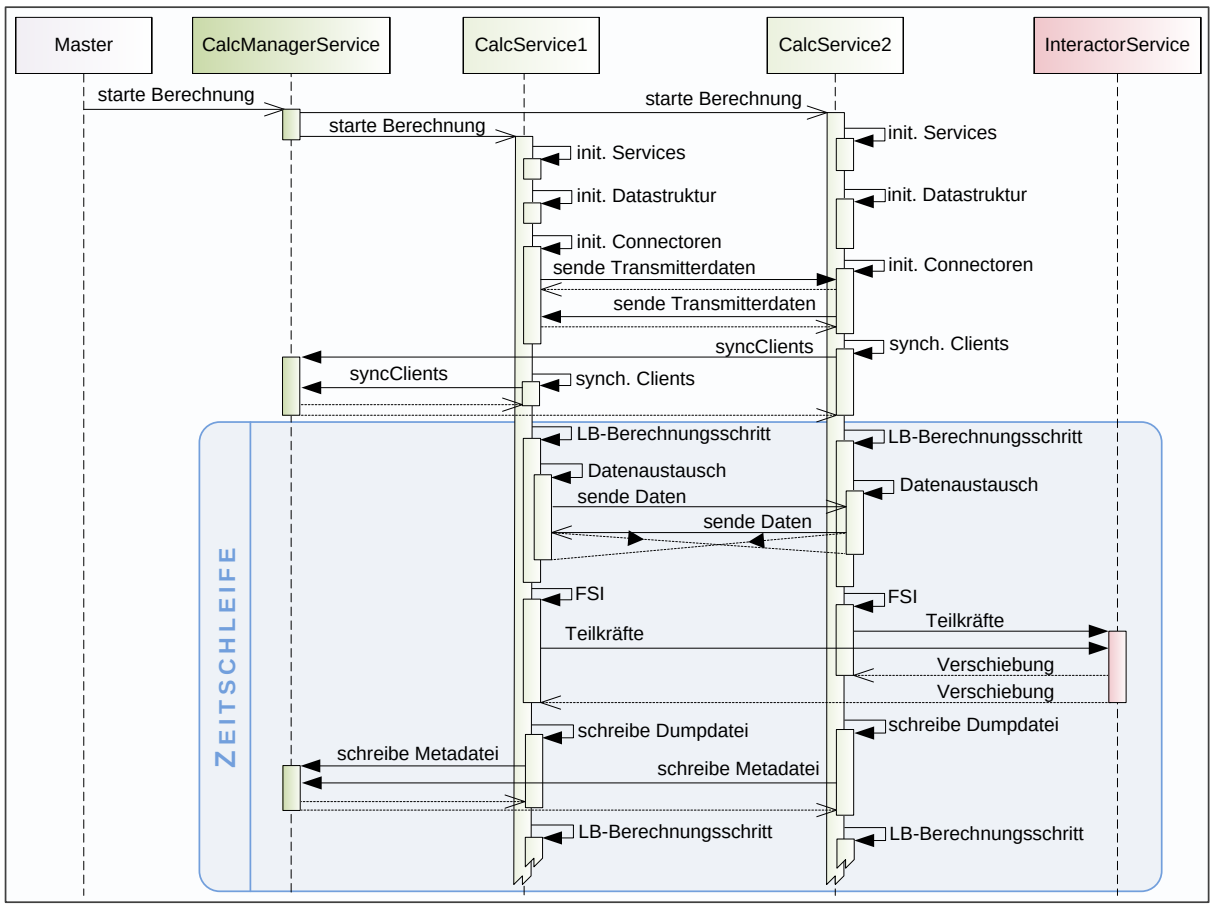


Abbildung 17.5: *Berechnungsroutine (UML-Sequenzdiagramm)*

Algorithmus 3 Berechnungsschleife

```
function TIMELOOP(start, stop)
    for s (=step) from start to stop do
        for ns (=nested step) from nestedStart to nestedStop do
            PREPARECONNECTORSFORRECEIVE( s, ns, startLevel[ns], stopLevel[ns] )
            SCALEFINETOCOARSE( s, ns, startLevel[ ns ], stopLevel[ ns ] )
            COLLIDEANDPROPAGATE( s, ns, startLevel[ ns ], stopLevel[ ns ] )
            EXCHANGESAMELEVELDATA( s, ns, startLevel[ ns ], stopLevel[ ns ] )
            APPLYBCS( s, ns, startLevel[ ns ], stopLevel[ ns ] )
            SCALECOARSETOFINE( s, ns, startLevel[ ns ], stopLevel[ ns ] )
            SWAPFS( s, ns, startLevel[ ns ], stopLevel[ ns ] )
            UPDATETIMEDEPENDENTVISITORS( s, ns )
            UPDATETIMEDEPENDENTINTERACTORS( s, ns )
            MOVEINTERACTORS( s, ns )
        end for
        PERFORMGRIDADAPTIVITY( s )
        NOTIFYOBSERVERS( s )
    end for
end function
```

Neben der LB-Berechnung finden in der Zeitschleife eine Reihe weiterer Methoden Anwendung, auf die in Abschn. 17.7 im Einzelnen eingegangen wird. So werden in updateTimedependentVisitors zeitabhängige Besucherklassen (z. B. für ein zeitlich variierendes Forcing) berücksichtigt. Dort werden u. a. kollektive Aufgaben bearbeitet, die Informationen aller Klienten benötigen (z. B. das Erstellen einer Ausgabemetadatei, die Berechnung von Kräften auf Festkörper, etc.). Die Anpassung zeitabhängiger Randbedingungsobjekte erfolgt in updateTimedependentInteractors und die Festkörperänderungen werden in moveInteractors ermittelt und umgesetzt. Außerhalb der inneren Zeitschleife erfolgt mit performGridAdaptivity die dynamische Gitteranpassung sowie die Benachrichtigung externer Beobachter mit notifyObservers.

17.7 Weitere Funktionalitäten der Services

17.7.1 Zentrale Bearbeitung kollektiver/verteilter Aufgaben

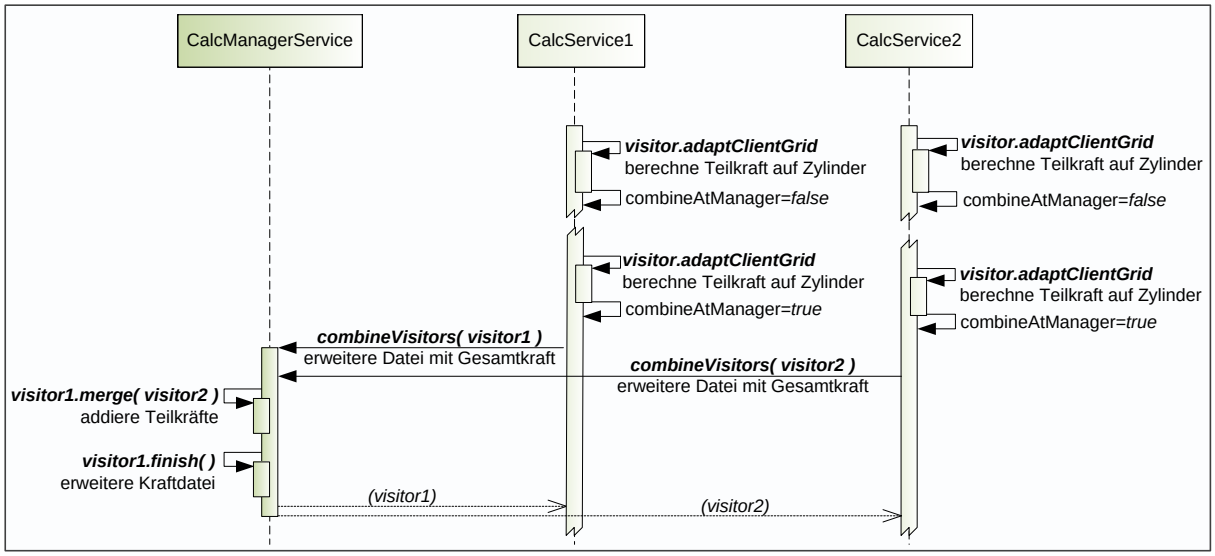


Abbildung 17.6: *ClientVisitor zur Bearbeitung kollektiver Aufgaben (UML-Sequenzdiagramm)*

Zum Bearbeiten von Aufgaben während der Berechnung können beliebige, zeitabhängige Besucherklassen verwendet werden. Ein Beispiel hierfür ist die Änderung physikalischer Eigenschaften (Viskosität, Volumenkraft, etc.).

Bei Fragestellungen, die nur durch die Kombination von verschiedenen Prozessinformationen bearbeitet werden können, wie die Berechnung der Gesamtkraft auf einen Körper, kommen ClientVisitor-Module zum Einsatz. Zu definierten Zeitpunkten wird auf jedem Prozess deren adaptClientGrid-Methode aufgerufen. Dort wird zum Beispiel die Teilkraft des Fluids auf einen Zylinder berechnet und gespeichert (Abb. 17.6). Anschließend werden die Besucherobjekte zum CalcManagerService übertragen und mit der im speziellen ClientVisitor definierten merge-Methode zusammengeführt. Der dabei auszuführende Algorithmus wird vom Anwender bestimmt. In diesem Beispiel werden die Teilkräfte auf einen Festkörper aufsummiert. Sobald alle Besuchermodule zusammengeführt wurden, wird die finish-Methode des resultierende ClientVisitor-Objekts aufgerufen. Diese erstellt bzw. erweitert hier die entsprechende Datei für den Postprozess um die ermittelten Kräfte.

Da sich Interprozesskommunikationen negativ auf die Gesamtperformance auswirken, kann der Anwender festlegen, ob ein Austausch mit dem CalcManagerService nach jedem lokalen adaptClientGrid-Aufruf erfolgen soll oder ob die Informationen zunächst in den Besucherobjekten gesammelt und nur nach bestimmten Zeitintervallen beim CalcManagerService zusammengeführt werden sollen. Die Erweiterung der Kraftdatei erfolgt in Abb. 17.6 in jedem zweiten Zeitschritt. Soll das ermittelte Ergebnis den Klienten verfügbar gemacht werden, so verteilt der CalcManagerService die Information vor der Rückübertragung auf die einzelnen Aufgabenmodule und sendet diese anschließend zu den CalcServices. Ein Anwendungsfall hierfür ist die Lokalisierung der auftretenden Maximalgeschwindigkeit des Gesamtsystems.

Auf der Seite der Rechendienste kann die gleichzeitige Ausführung von Operationen in adaptClientGrid beschränkt werden. Dies ist beispielsweise sinnvoll, wenn die CalcServices ihre Teilgitter auf einen zentralen Server ohne paralleles Dateisystem schreiben sollen. Für eine optimale Effizienz wird dort die Anzahl der gleichzeitig schreibenden Prozesse auf eine benutzerdefinierte Anzahl beschränkt.

Für den Fall, dass mehrere Aufgaben zum gleichen Zeitpunkt bearbeitet werden sollen, werden diese gemeinsam in einem Aufruf und somit zeitsparend zum CalcManagerService übertragen. Die Unterstützung von *inout*-Argumenten seitens RCF ermöglicht hierbei die einfache Rückübertragung der Ergebnisobjekte.

17.7.2 Bearbeitung externer Anfragen

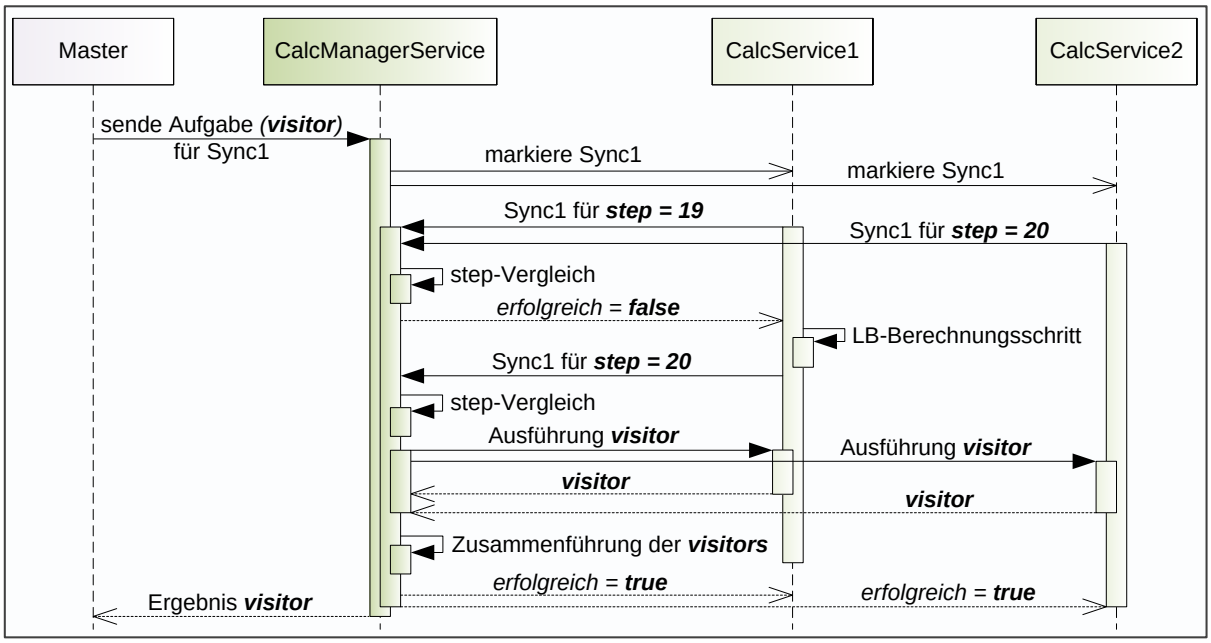


Abbildung 17.7: *Bearbeitung externer Anfragen (UML-Sequenzdiagramm)*

Die Bearbeitung externer Anfragen erfolgt auf ähnliche Weise wie in Abs. 17.7.1. Diese Schnittstelle ist u. a. für die Anbindung einer externen GUI vorgesehen. Eine oder mehrere zu bearbeitende Anfragen/Aufgaben werden in Form von Besucherobjekten direkt an den CalcManagerService gesendet. Um zu berücksichtigen, dass Aufgaben eventuell an unterschiedlichen Positionen der Zeitschleife bearbeitet werden sollen, muss der Aufrufer den entsprechenden Synchronisierungspunkt angeben.

Der CalcManagerService informiert zunächst die Rechenprozesse darüber, dass externe Aufgaben vorliegen. Dazu wird vom CalcService der gewünschte Synchronisierungspunkt in der Berechnungsklasse markiert. Sobald dieser zur Laufzeit erreicht wird, verbinden sich die Rechendienste mit dem CalcManagerService. Zur Bewahrung der Konsistenz wartet dieser solange, bis sich alle Klienten im gleichen Zeitschritt befinden (vgl. Abb. 17.7). Anschließend werden die Aufgaben an die CalcServices übertragen und dort bearbeitet. Handelt es sich um informationsammelnde Module, so werden diese beim CalcManagerService zusammengeführt und anschließend an den externen Aufrufer übermittelt.

17.7.3 Publisher-Subscriber-Dienst

Der CalcManagerService stellt beim Start einen *Publisher*-Server zur Verfügung, bei dem sich z. B. externe Anwendungen als *Subscriber* eintragen können. Alle Nachrichten, die über den *Publisher*-Server versendet werden, leitet dieser anschließend zu den registrierten *Subscribern* weiter. Diese ein bestimmtes Interface implementierende Nachrichten können dort unterschiedlich behandelt werden. Derzeit können auf diesem Weg z. B. der aktuelle Berechnungsschritt der Simulation oder Fehlermeldungen übermittelt werden.

17.7.4 Fluid-Struktur-Interaktion (FSI)

Um bewegliche und sich verformende Geometrien zu berücksichtigen, werden in VIRTUALFLUIDS sogenannte SteeringPlugins verwendet (vgl. Abs. 16.3.2). Diese enthalten die Vorschrift, wie sich eine Geometrie zur Laufzeit verhält. Im einfachsten Fall verschiebt ein solches Modul den Festkörper in jedem Zeitschritt um einen konstanten Wert. Weitaus komplexer sind Plugins, die an externe Strukturlöser, wie AdhoC [32] oder die PHYSICSENGINE (pe) von Klaus Iglberger [57, 81], gekoppelt sind.

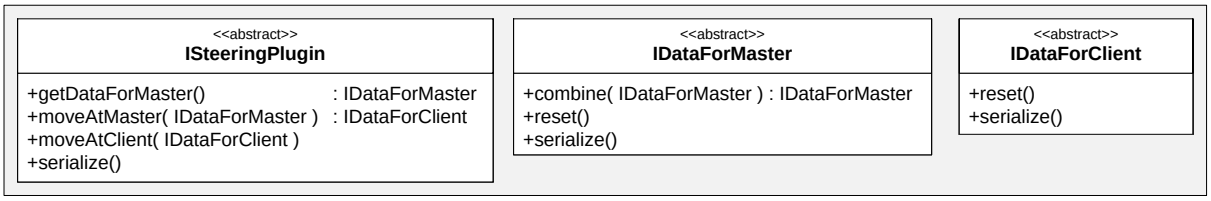


Abbildung 17.8: *SteeringPlugin-Schnittstelle (UML)*

Im Folgenden wird die Umsetzung dieser Steuerungsmodule für verteilte Löser erläutert. Das konkrete Plugin implementiert die Schnittstelle ISteeringPlugin (Abb. 17.8). Zudem existieren mit IDataForMaster und IDataForClient zwei Hilfsschnittstellen, deren Implementierungen als Datencontainer fungieren, die später zwischen den Diensten ausgetauscht werden. In der Anwendung wird die Fluid-Struktur-Interaktion (FSI) in der Methode moveInteractors der allgemeinen Berechnungsklasse durchgeführt (Abschn. 17.6: Alg. 3). Jeder Rechenprozess ermittelt dort zuerst mit getDataForMaster der SteeringPlugin-Klasse die vom Strukturlöser benötigten Teildaten (z. B. anteilige, schwerpunktbezogene Kräfte aus dem Fluid auf die Struktur) und speichert sie in einem IDataForMaster-kompatiblen Container. Dieser wird im Anschluss an den InteractorService gesendet und dort mit denen der anderen Dienste kombiniert (Abb. 17.9). Hierfür kommt die combine-Methode der Hilfsklasse zum Einsatz, in der beispielsweise die Addition der Teilkräfte erfolgt.

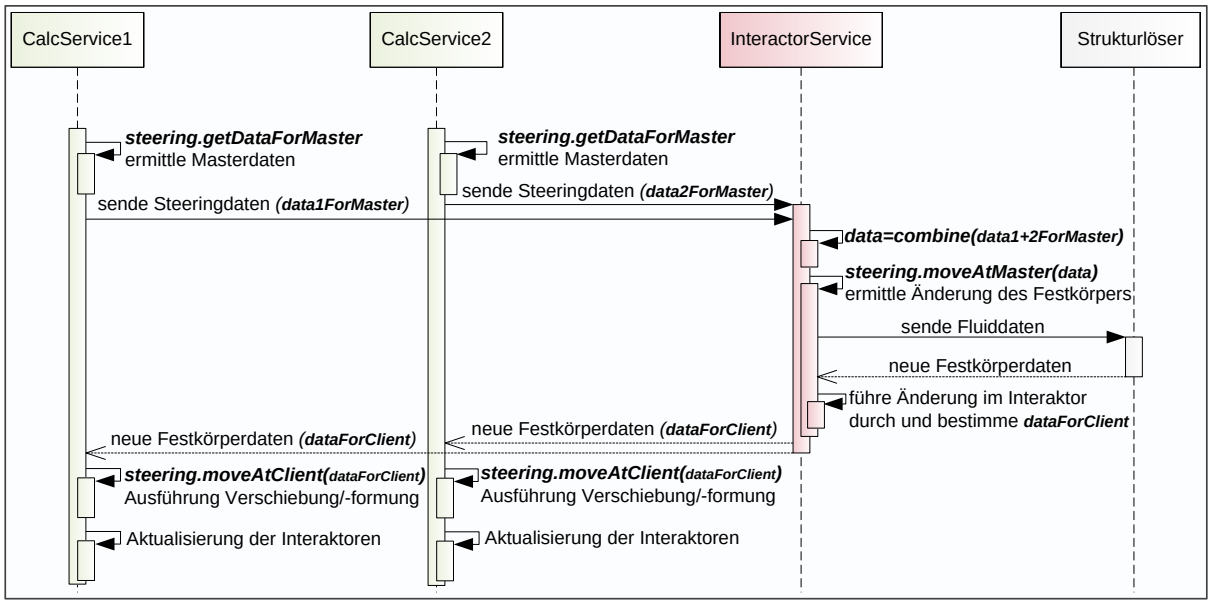
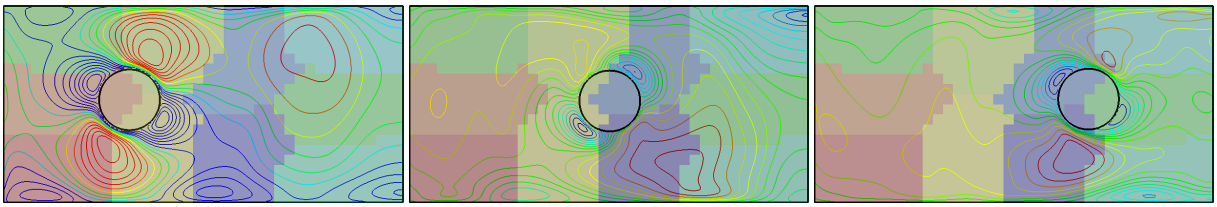


Abbildung 17.9: *Fluid-Struktur-Interaktion (UML-Sequenzdiagramm)*

Anschließend sind die für die Änderung der Struktur notwendigen Daten vollständig. Unter deren Verwendung erfolgt in moveAtMaster des SteeringPlugins als nächstes die Interaktion mit dem Geometrieobjekt, die je nach Modul durch die Kopplung mit einem Strukturlöser durchgeführt wird. Sobald die neuen Geometriedaten bestimmt wurden, wird der lokale Interaktor und somit die Geometrie angepasst. Die für die Rechendienste notwendigen Änderungen werden dabei in einem IDataForClient-Objekt gespeichert. Nachdem dieses zu den CalcServices geschickt wurde, führen die Rechenprozesse mit diesen Informationen die entsprechende lokale Änderung der Struktur durch. Nach der im Anschluss durchgeführten Aktualisierung der Interaktoren kann die LB-Berechnung fortgesetzt werden.

Als Beispiel ist in Abb. 17.10 ein sich durch einen Kanal bewegender Zylinder dargestellt. Die Isolinien entsprechen der u_{x_1} Komponente, und im Hintergrund ist die statische METIS-Segmentierung dargestellt.



(a) *Berechnungsschritt 4.000* ***(b)*** *Berechnungsschritt 12.000* ***(c)*** *Berechnungsschritt 16.000*

Abbildung 17.10: *Mittels FSI bewegter Zylinder im Kanal auf einem verteilten Gitter (2-D)*

Im Prototypen werden, um die Implementierung einfach zu halten, *solid* Blöcke bei der Gebietszerlegung und der Zuweisung der Connectoren wie *fluid* Blöcke behandelt. Dies hat zum Vorteil, dass zur Laufzeit keine eventuell fehlenden Blöcke und Connectoren initialisiert werden müssen. Auf der anderen Seite kann dies unter Umständen eine unausgewogene Lastverteilung zur Folge haben.

17.7.5 Verteilte Adaptivitätssteuerung

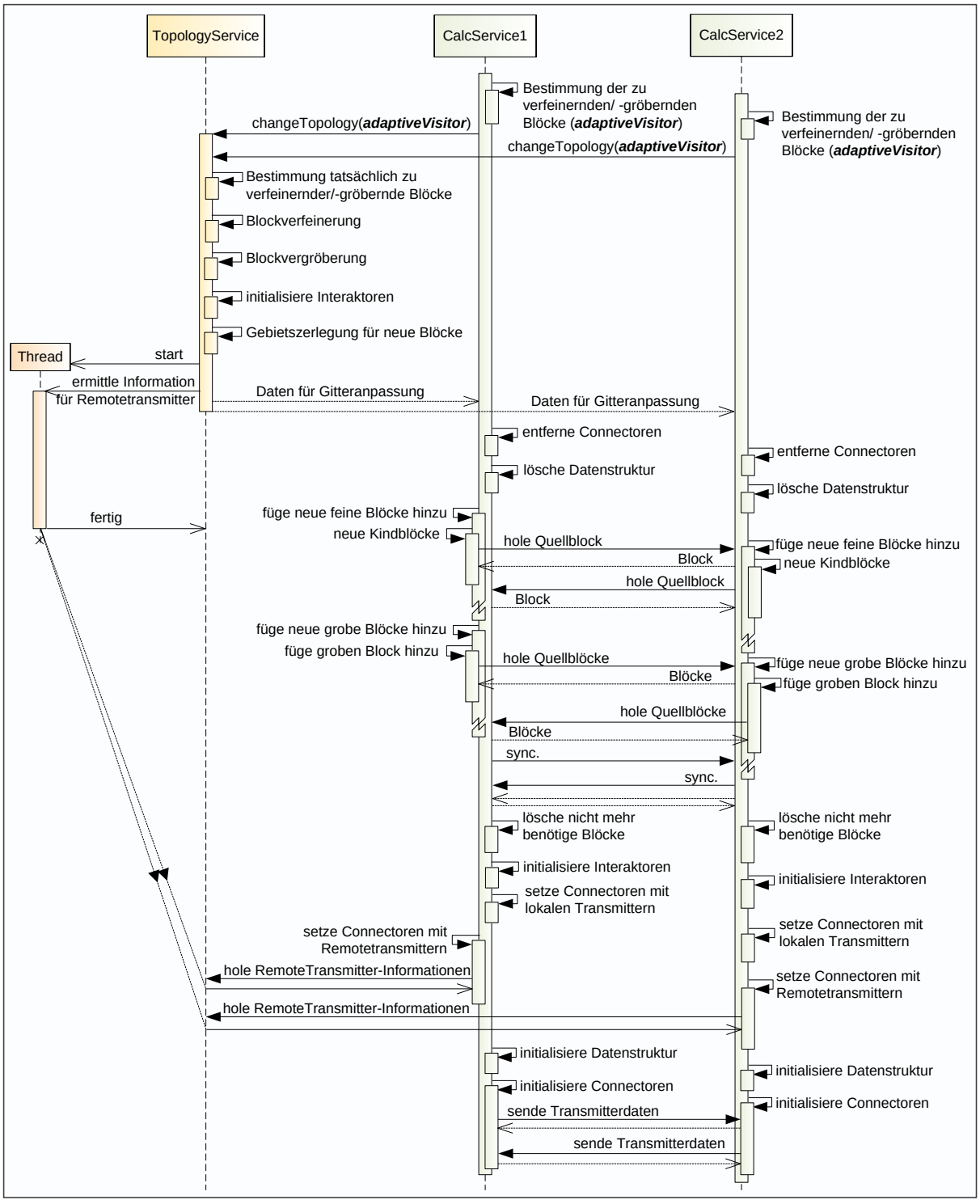


Abbildung 17.11: *Gitteradaptivität (UML-Sequenzdiagramm)*

Die schematische Vorgehensweise bei der dynamischen Gitteranpassung bei verteilten Gittern ist in Abb. 17.11 abgebildet. Zum besseren Verständnis wird das komplexe Schema im Folgenden, wie be-

reits für den Knotencode, anhand einer aufsteigenden Blase im Detail beschrieben. Die Ausgangstopologie ist in Abb. 17.12 in Form der anzupassenden Blockstruktur (ohne Knoten) für zwei Rechendienste abgebildet.

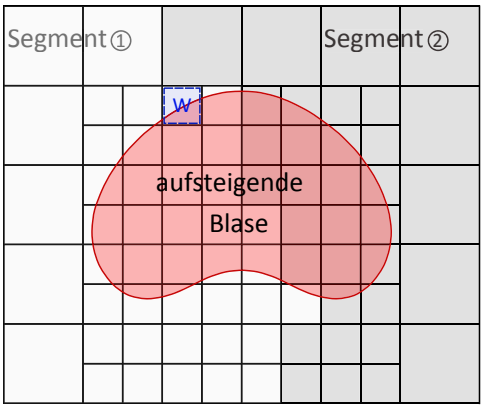


Abbildung 17.12: *Blockgitterausschnitt mit zwei Segmenten*

17.7.5.1 Bestimmung der anzupassenden Blöcke auf den Rechendiensten

Die Gitteranpassung erfolgt im Blockcode, wie im Knotencode, ausschließlich innerhalb des gröbsten Zeitschrittes (vgl. Abschn. 8.1). Zu Beginn der dynamischen Anpassung ermittelt jeder Rechenprozess mit entsprechenden Besuchermodulen die zu verfeinernden und zu vergröbernden Blöcke. Im Mehrphasenkontext werden hierzu die einzelnen Knoten der Blöcke im Bereich des Phaseninterfaces untersucht. Dabei werden die Blöcke, die keinen Interfaceknoten besitzen, in die Liste der zu vergröbernden Blöcke aufgenommen.

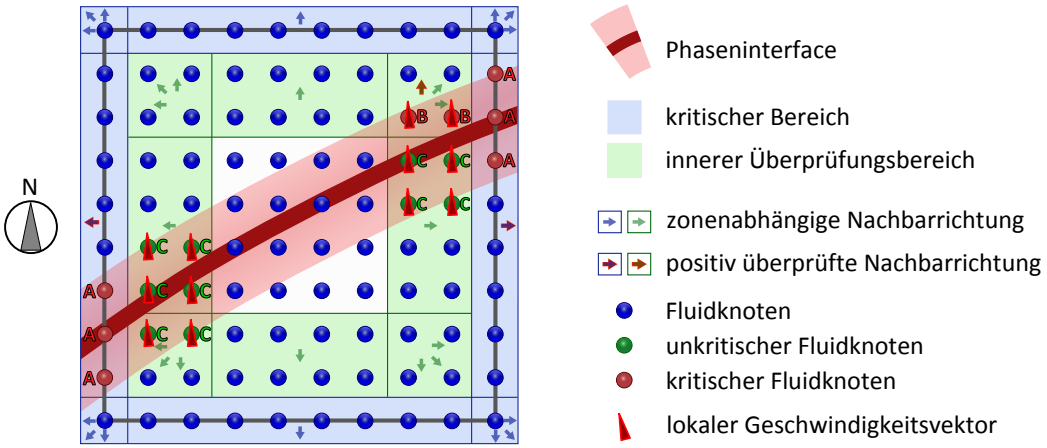
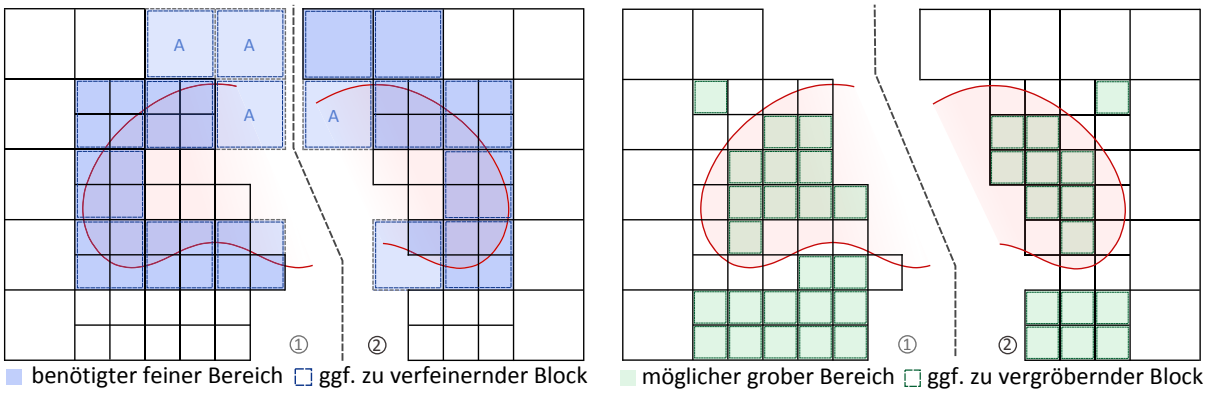


Abbildung 17.13: *Knotenmatrix von Block W*

Für die übrigen wird untersucht, ob einer seiner Nachbarblöcke den feinsten Level besitzen muss. Hierbei sind zwei frei definierbare Überprüfungsbereiche innerhalb der Knotenmatrix von Bedeutung (Abb. 17.13). Befindet sich ein Interfaceknoten im *kritischen Bereich*, so muss der Nachbarblock in der *zonenabhängigen Nachbarrichtung* die feinste Auflösung aufweisen. In dem dargestellten Knotenblock betrifft dies aufgrund der Interfaceknoten A sowohl den östlichen als auch den westlichen

Nachbarn. Da sich die angrenzenden Blöcke bei verteilten Berechnungen u. U. auf einem anderen Prozess befinden, ist eine direkte Überprüfung oftmals nicht möglich. Aus diesem Grund werden die Elternblockindizes des notwendigen Nachbarblocks gespeichert, der bei Vorhandensein später vom TopologyService verfeinert werden muss.

Im inneren Überprüfungsbereich wird unter Einbezug der lokalen Knotengeschwindigkeit die Bewegung des Phaseninterfaces berücksichtigt. Sobald sich eine Geschwindigkeitskomponente in zonenabhängiger Nachbarrichtung oberhalb eines definierbaren Schwellwertes befindet, wird, wie zuvor, der entsprechende Elternblock in die Liste der zu verfeinernden Blöcke hinzugefügt. Durch diese Vorgehensweise wird die Anzahl der Freiheitsgrade minimiert, da nur Blöcke in Fließrichtung der Phase verfeinert werden und der Nachlauf der Blase frühzeitig vergröbert werden kann. Im Beispiel muss der nördliche Nachbar ggf. verfeinert werden, denn die B-Knoten überschreiten den gesetzten Grenzwert in dieser Richtung. Die Geschwindigkeitsanteile der C-Knoten hingegen liegen für die einzubeziehenden Richtungen unter dem Schwellwert und sind somit unkritisch.



(a) *Notwendige feine Bereiche* ***(b)*** *Zu vergröbernde Bereiche*

Abbildung 17.14: *Ermittelte Anpassungsbereiche*

Sobald alle Blöcke untersucht wurden, werden die gesammelten Daten an den TopologyService geschickt. In Abb. 17.14 sind die ermittelten Informationen für die aufsteigende Blase abgebildet. Hierbei ist zu erwähnen, dass der benötigte feine Bereich in Abb. 17.14a ausschließlich mit der beschriebenen Nachbarschaftsmarkierung ermittelt wurde. Auch wenn diese Blöcke bereits im verfeinerten Zustand vorliegen, benötigt der TopologyService diese Daten zur Identifizierung der tatsächlich anzupassenden Elemente. Die A-Blöcke markieren dabei benötigte Zonen, die in einem anderen Segment liegen und eventuell verfeinert werden müssen.

17.7.5.2 Anpassung des Blockgitters auf dem TopologyService

Nachdem der TopologyService die Informationen von den Rechendiensten empfangen hat, werden diese im ersten Schritt miteinander kombiniert. Dabei werden u. a. die Blöcke aussortiert, die vergröbert werden sollen, sich aber in Bereichen befinden, die später den feinsten Level aufweisen sollen. Zur Bewahrung eines konsistenten Quad-/Octrees müssen beim Erstellen eines Elternblocks *alle* zugehörigen Kindzellen vergröbert werden. Dies gewährleistet, dass ausschließlich Knoten des Quad-/Octrees Blätter als Nachfolger besitzen (vgl. Abb. 17.15). Aus diesem Grund werden auch die zu vergröbernden Blöcke aussortiert, die mindestens einen zum gleichen Elternblock gehörenden Nachbarn besitzen, der nicht vergröbert werden soll.

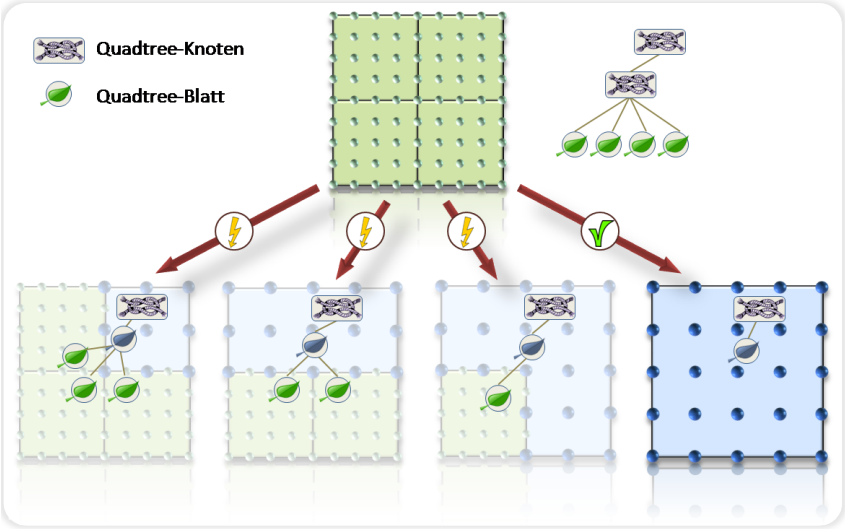
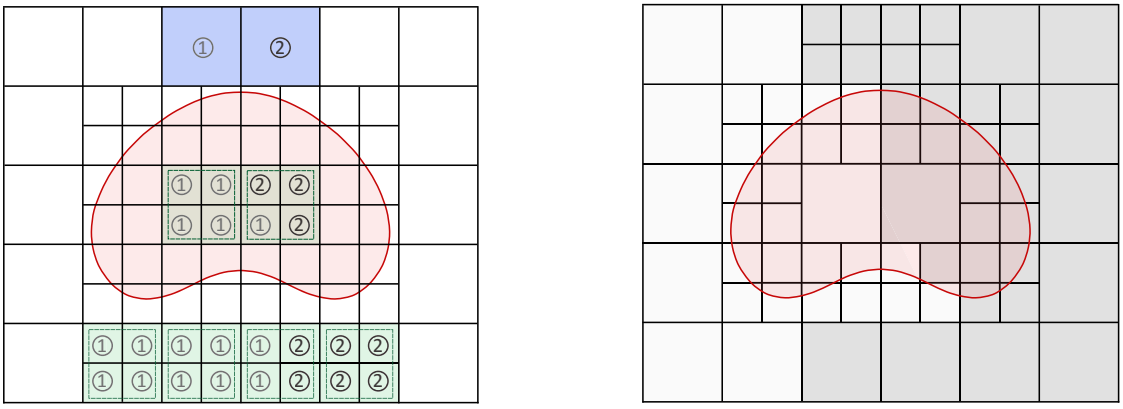


Abbildung 17.15: *Zulässige Blockvergröberung*

Als letztes werden die zu verfeinernden Blöcke aus den Listen entfernt, die entweder nicht existieren oder einen Bereich markieren, der sich bereits im feinsten Level befindet. Somit ergibt sich für das Beispiel die in Abb. 17.16a dargestellte Anpassungsvorschrift, die nun vom TopologyService umgesetzt wird. Für die spätere Datenzuweisung auf den Rechenprozessen werden dabei die Quellblöcke in Relation zu den neu generierten Blöcken sowie die zugehörigen Prozessnummern gespeichert. Nach erfolgreicher Topologieänderung (Abb. 17.16b) werden die Interaktoren initialisiert, um für die anschließende Gebietszerlegung ein aktualisiertes, gültiges Gitter zu erhalten.



(a) *Zusammengeführte Informationen* ***(b)*** *Angepasstes Blockgitter*

Abbildung 17.16: *Topologieänderung des Blockgitters*

Da METIS keine adaptive Graphenänderungen unterstützt, wird im Prototypen vorerst eine heuristische Neuverteilung verwendet. Hierzu wird der neu zu verteilende Arbeitsaufwand ermittelt und so auf die bestehenden Klienten verteilt, dass diese später den gleichen Rechenaufwand besitzen. Bevor dem jeweiligen Dienst die ihn betreffende Gitteränderung sowie die für die Datenzuweisungen notwendigen Informationen übertragen werden, startet der TopologyService in einem asynchronen Thread die Ermittlung der RemoteTransmitter-Informationen. Dadurch können die CalcServices parallel dazu mit der Anpassung der Teilgitter beginnen.

17.7.5.3 Anpassung des Blockgitters auf den CalcServices

Nach erfolgter Gitteranpassung auf dem TopologyService wird zunächst die nicht mehr aktuelle Datenstruktur der Berechnungsklasse zurückgesetzt und im Anschluss die neuen Blöcke dem Gitter hinzugefügt.

Für die physikalische Wertzuweisung der neuen Blöcke werden die zugehörigen Quellblöcke benötigt. Die hierfür notwendigen Relationen und Prozessinformationen wurden vom TopologyService ermittelt und an die Klienten weitergeleitet. Ist ein erforderlicher Block lokal nicht verfügbar, wird er über das RCF vom entsprechenden Dienst angefordert. Hierbei kommt die Flexibilität des RCF zur Geltung. Die Serviceanfragen können in beliebiger Reihenfolge durchgeführt werden, und der Programmierer muss sich nicht um die Umwandlung der Objekte in ein über das Netzwerk versendbares Datenformat kümmern. Diese Aufgabe übernimmt das Serialisierungs-Framework SF von Jarl Lindrud oder optional das der Boost-Bibliothek (vgl. Abs. A5.3). Ein weiterer Vorteil ist, dass die bezogenen Objekte strukturell identisch mit den lokal erzeugten sind und somit die Funktionen für die Wertzuweisung unverändert verwendet werden können. Die Interpolation zwischen den Gitterleveln erfolgt nach dem bereits beim Knotengitter angewandten Schema (vgl. Abschn. 8.3). Um zu vermeiden, dass für die Wertzuweisung anderer Prozesse noch benötigte Blockdaten vorzeitig gelöscht werden, entfernen die Rechendienste diese Daten erst nachdem alle CalcServices die Topologieänderung durchgeführt und eine Synchronisierung durchgeführt haben.

Im Anschluss werden den Knoten die benötigten Randbedingungsflags und den Blöcken die Connector- und Übertragungsmodule zugewiesen. Hierzu werden die Interaktoren aktualisiert und die entsprechenden Besuchermodule angewendet. Sobald die für die Zuordnung der RemoteTransmitter benötigten Informationen auf dem TopologyService vorliegen, beziehen die CalcServices diese und erstellen die noch fehlenden Connectoren und RemoteTransmitter. Sobald die Verbindungsmodule initialisiert und die restliche Datenstruktur aktualisiert wurde, erfolgt der nächste LB-Berechnungsschritt.

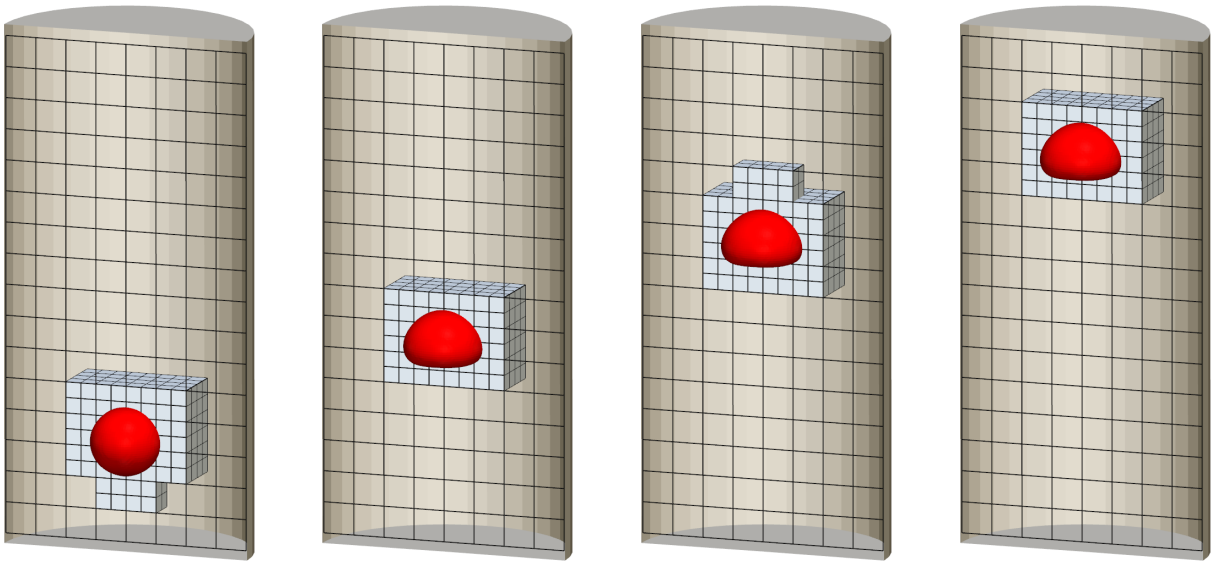


Abbildung 17.17: *Aufsteigende Blase mit dynamisch adaptiven Blockgitter*

In Abb. 17.17 ist beispielhaft eine aufsteigende Blase mit adaptiver Gitterstruktur dargestellt. Für rein serielle Anwendungen ohne Verwendung der Dienste erfolgt die Anpassung mit denselben Modulen in vergleichbarer Reihenfolge.

17.8 Optimierungen

17.8.1 Parallele Bearbeitung von Anfragen an die CalcServices

Anfragen an den CalcManagerService wurden in den ersten Versionen von VIRTUALFLUIDS mit verteilten Blockgittern sequenziell und prozessblockierend an die Rechendienste weitergeleitet (Abb. 17.18).

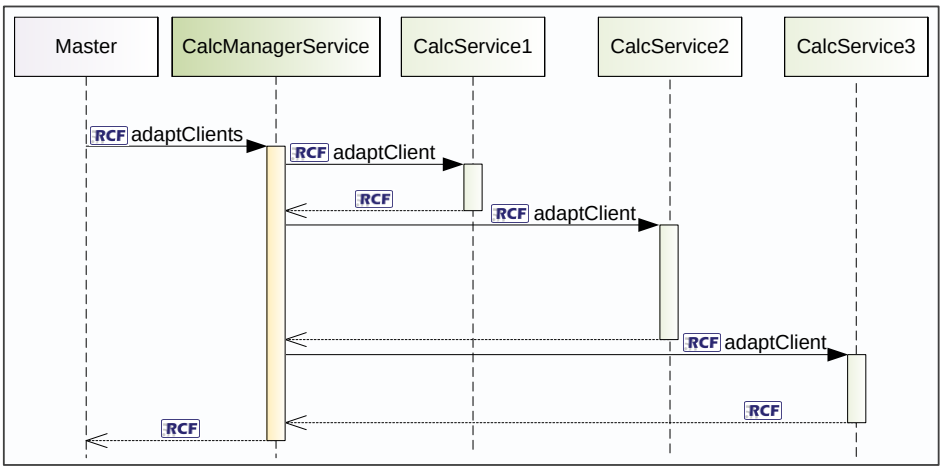


Abbildung 17.18: *Ausführung verteilter Aufgaben (Singlethreaded) (UML-Sequenzdiagramm)*

Dadurch wurden diese von jeweils nur einem CalcService zur gleichen Zeit bearbeitet. Diese Vorgehensweise ist insbesondere bei einer hohen Prozessanzahl sehr ineffizient, da z. B. die Initialisierung des Strömungsfeldes von allen Rechenclients unabhängig voneinander zum gleichen Zeitpunkt durchgeführt werden kann. Eine mögliche Lösung sind nicht blockierende Einwegaufrufe seitens des CalcManagerServices. Das Problem hierbei ist jedoch, dass der aufrufende Dienst dabei keine Antwort vom bearbeitenden Service erhält und somit die Gefahr von Überschneidung der ausgeführten Befehle besteht. Aus diesem Grund wurde der Dienst dahingehend erweitert, dass die Befehle optional mit Hilfe von Threads zeitlich parallel an die Klienten versendet und dort bearbeitet werden (Abb. 17.19).

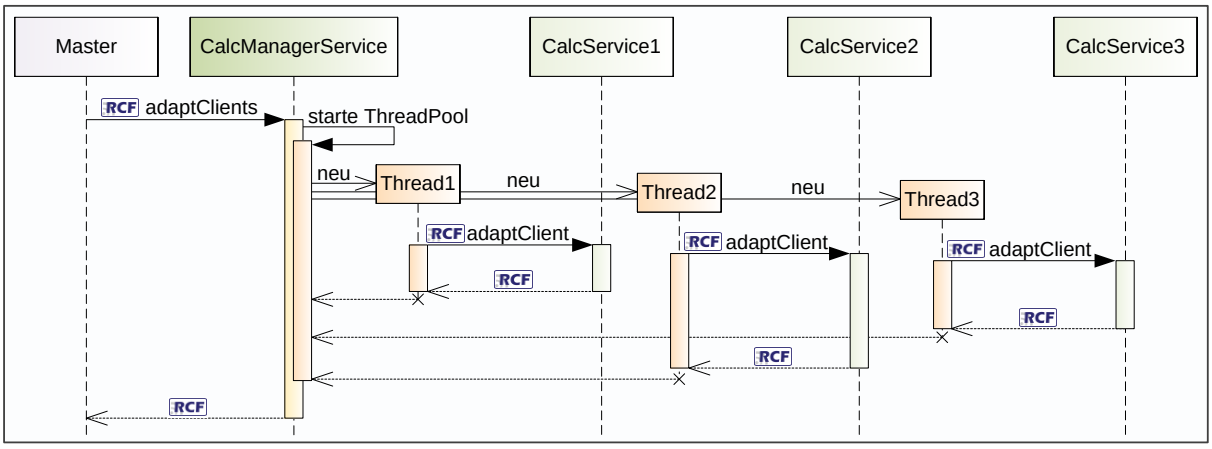


Abbildung 17.19: *Ausführung verteilter Aufgaben (Multithreaded) (UML-Sequenzdiagramm)*

Der CalcManagerService wartet anschließend, bis alle Threads beendet sind und somit alle Aufgaben bearbeitet wurden. Anschließend kehrt er zum aufrufenden Prozess zurück. Für datenintensive, netzwerkbelastende Anfragen können die Aufrufe optional zeitlich verzögert an die einzelnen Dienste weitergeleitet werden.

17.8.2 Optimierung des Datenaustausches zur Berechnungslaufzeit

Um den Vorteil der asynchronen Datenübertragung nutzen zu können, wurden die Austauschroutinen entsprechend angepasst. In der ersten Version wurden die von den Connectoren benötigten Daten in beliebiger Reihenfolge und ohne Berücksichtigung des jeweiligen Transmittertyps gesammelt, gesendet, empfangen und anschließend einsortiert (Alg. 4).

Algorithmus 4 Connectordatenaustausch (alt)

```
function EXCHANGEDATA( )
    for all connectors do
        connector.collectData()
        connector.sendData()
        connector.receiveData()
        connector.distributeData()
    end for
end function
```

Algorithmus 5 Connectordatenaustausch (neu)

```
function EXCHANGEDATA( )
    for all remoteConnectors do
        remoteConnector.prepareForReceive()
        remoteConnector.collectData()
        remoteConnector.sendData()
    end for
    for all localConnectors do
        localConnector.collectData()
        localConnector.sendData()
        localConnector.receiveData()
        localConnector.distributeData()
    end for
    for all remoteConnectors do
        remoteConnector.receiveData()
        remoteConnector.distributeData()
    end for
end function
```

In der optimierten Version werden im Präprozess beim Erstellen der Datenstruktur die lokalen Connectoren, die ausschließlich lokal operieren, von den RemoteTransmittern enthaltenen Connectoren getrennt. Dadurch können diesen bei der Blockkommunikation separat behandelt werden. Als erstes ermitteln die RemoteConnectoren die notwendigen Daten und senden diese anschließend an die jeweiligen Nachbarprozesse. Im optimalen Fall kann der Sendevorgang im Hintergrund erfolgen, sodass in der Zwischenzeit die Abarbeitung der ausschließlich lokal fungierenden Connectoren erfolgen kann. Erst im Anschluss werden die im Hintergrund von den anderen Prozessen empfangenen Daten auf die Blöcke verteilt (Alg. 5).

Einige RemoteTransmitter müssen vor dem Datenempfang in Empfangsbereitschaft gesetzt werden, um zu gewährleisten, dass noch benötigte Datenpuffer nicht vorzeitig überschrieben werden. Dies erfolgt in der Transmittermethode prepareForReceive, die entweder in exchangeData der Berechnungsklasse (vgl. Alg. 5) oder separat zu Beginn eines jeden Zeitschrittes aufgerufen wird. Bei MPI wird die gewünschte Funktionalität in diesem Zusammenhang z. B. durch die Verwendung MPI-spezifischer *request*-Module erreicht. Als weitere Optimierungsmaßnahme werden die Connectoren im Präprozess entsprechend ihrer Zielprozessnummer sortiert.

17.8.3 Optimierung der RemoteTransmitter

Ein entscheidender Faktor für die Gesamtperformance ist der relative Zeitbedarf der Datenübermittlung während eines Zeitschritts. Neben der Optimierung der Übertragungsalgorithmen und Ausnutzung der Netzwerkbandbreite muss hier auch der Anteil der Latenzzeit, also die Zeit, die ausschließlich für den Aufbau einer Verbindung zwischen zwei Prozessen benötigt wird, minimiert werden. Das Connector-Transmitter-Konzept sieht eine Übertragung pro Nachbarrichtung und Zeitschritt vor. Übertragen nur wenige Blöcke viele Daten zum Nachbarprozess, so ist der Einfluss der dadurch entstehenden Latenzzeit vernachlässigbar. Für den umgekehrten Fall kann es zu hohen Performanceeinbußen kommen.

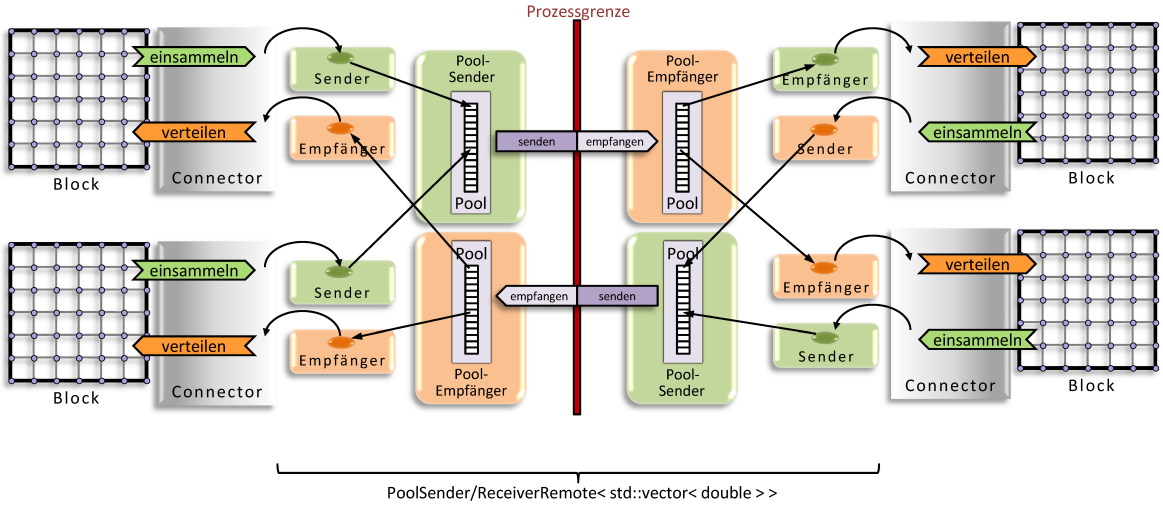


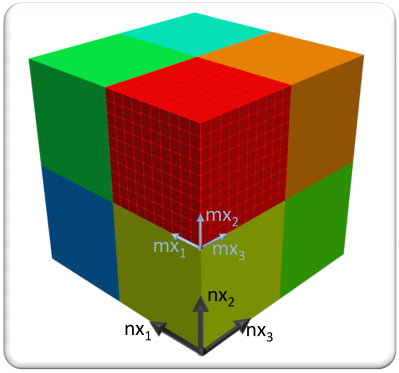
Abbildung 17.20: *D3Q19VectorConnector mit lokalem Vektorpooltransmitter*

Um dieses Defizit zu vermeiden, wurde der Vektorpooltransmitter entwickelt (Abb. 17.20). Dessen Grundprinzip besteht darin, die Transmitter zu gruppieren, die zum gleichen Zeitpunkt Daten mit demselben Prozess austauschen müssen. Die Transmitter besitzen dabei kein eigenes Datenfeld, sondern referenzieren auf ein ihnen zugewiesenes Segment in einem kontinuierlichen Speicherbereich, der in einem Vektorpool vorgehalten wird. Dieser wird vom PoolSender zum Austauschzeitpunkt in einem Schritt zum PoolEmpfänger des Nachbarprozesses übertragen. Anschließend stehen dort den Empfangsmodulen die Informationen zur Verfügung. Die vom Vektorpool bereitgestellten Teilvektoren verfügen über die notwendigen Basisfunktionen der STL-Vektoren, wie z. B. den Indizierungsoperator *[]* oder die size- und resize-Methode und sind somit vollständig connectorkompatibel.

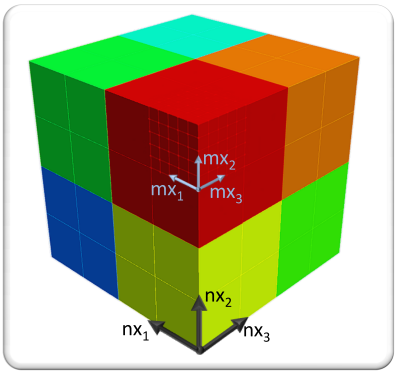
Um den Einfluss des gewählten Transmittertyps auf die Effizienz zu ermitteln, wurde mit dem 3-D-Code ein einfacher Testfall ohne Festkörper unter Verwendung des inkompressiblen BGK-Modells (vgl. Abschn. 3.1) berechnet. Dafür wurde ein kubisches, in alle Richtungen periodisches Gebiet mit $nx_1 \cdot nx_2 \cdot nx_3$ Blöcken und jeweils $mx_1 \cdot mx_2 \cdot mx_3$ Knoten pro Block in äquivalente Teilgebiete unterteilt (Abb. 17.21). Für diese sind sowohl der Rechenaufwand als auch der lokale und entfernte Kommunikationsaufwand identisch.

Verglichen wurden zwei verschiedene Gitterkonfigurationen für unterschiedliche Transmittertypen. Bei der ersten befand sich auf jedem Clusterknoten genau ein Block mit 100^3 Berechnungsknoten (vgl. Abb. 17.21a). Die Blockkommunikation fand dabei mit Ausnahme des seriellen Referenztestfalls für jede Richtung zwischen zwei unterschiedlichen Prozessen statt. Um das Verhalten bei einer hohen Anzahl an Einzelblockkommunikationen pro Zeitschritt zu untersuchen, wurde der einzelne Block im

zweiten Setup in 10^3 Blöcke mit jeweils 10^3 Knoten unterteilt (vgl. Abb. 17.21b). Für die lokale Interblockkommunikation kamen DirectConnectoren (vgl. Abschn. 15.2) zum Einsatz, wodurch in diesem Beispiel gegenüber der Verwendung von VectorConnectoren ca. 5 - 8 % mehr Leistung erzielt werden konnte.

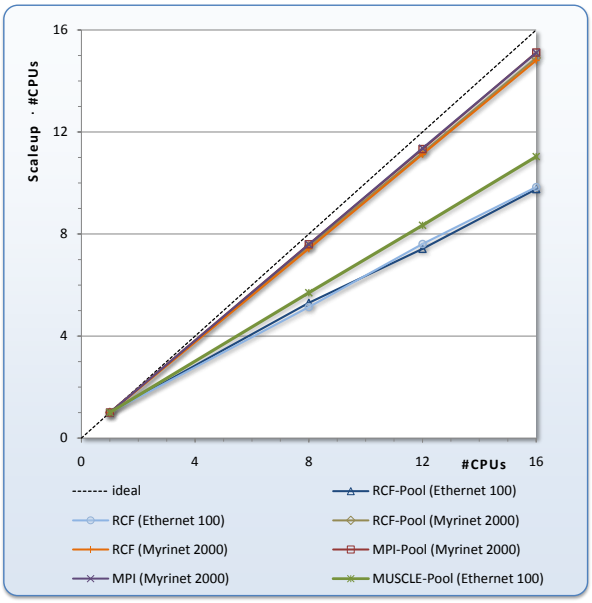


(a) *(2 · 2 · 2) - (10 · 10 · 10)*

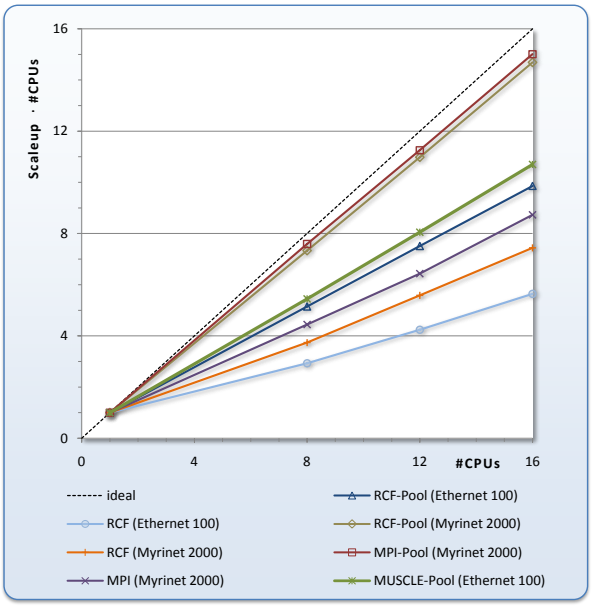


(b) *(4 · 4 · 4) - (5 · 5 · 5)*

Abbildung 17.21: *Transmittertestsetup $(nx_1 \cdot nx_2 \cdot nx_3)$ - $(mx_1 \cdot mx_2 \cdot mx_3)$*



(a) *Ein Block mit je 100^3 Knoten pro Client*



(b) *10^3 Blöcke mit je 10^3 Knoten pro Client*

Abbildung 17.22: *3-D-Transmittertest: Scaleup*

Das zugrunde liegende Testsystem für die Berechnungen war auch hier der iRMB-PC-Cluster (vgl. Abschn. 17.9). Wie in Abb. 17.22a zu erkennen, ist für den ersten Testfall der lineare Scaleup der Vektorpooltransmitter mit dem der Einzeltransmitter identisch. Nachteile durch die Verwendung des Vektorpools, wie z. B. verschlechterte Zugriffszeiten auf die Daten, sind somit ausgeschlossen. Bei der RCF-Implementierung wurden zum einen die verhältnismäßig langsame *Ethernet 100*- und zum anderen die vollduplexfähige *Myrinet 2000*-Schnittstelle verwendet. In beiden Fällen kam das TCP-Protokoll zum Einsatz. Wie erwartet, wurden für das Myrinetnetzwerk die besseren Leistungswerte erzielt. Diese unterschieden sich zudem nur geringfügig von denen, die mit Transmittern erzielt

wurden, die auf einer nativen, speziell für das Myrinet optimierten MPI-1-Implementierung basierten.

Im Gegensatz zum ersten Testfall, bei dem pro Prozess und Zeitschritt je 36 Interprozessübertragungen (18 Sender + 18 Empfänger) durchgeführt wurden, waren es im zweiten 5.760. Hinzu kamen weitere 30.240 lokale Transferoperationen. Die Verwendung der Einzeltransmitter bei vielen Blöcken mit kleinen Knotenmatrizen und somit vielen Interprozessaufrufen pro Zeitschritt führt erwartungsgemäß zu einer ausgeprägten Leistungsreduzierung (Abb. 17.22b). Ursache hierfür ist die Aufsummierung der bei jedem Netzwerkaufruf auftretenden Latenzzeit (Gl. 17.1). Diese entspricht der Zeit, die eine Nachricht für den Weg vom Sender bis zum Empfänger benötigt und umfasst sowohl die Ausbreitungs-, Übertragungsverzögerung und die ggf. anfallende Wartezeit.

$$t_{Kommunikation}[s] = t_{Latenz}[s] + \frac{Transfervolumen\,[MBit]}{Bandbreite\,[MBit/s]} \tag{17.1}$$

Des Weiteren ist beim Vergleich der Myrinet nutzenden Einzeltransmitter festzustellen, dass die RCF-Module nicht die Leistung der MPI-Transmitter erreichen. Mögliche Ursache hierfür ist der Mehraufwand, der durch die Verwendung des TCP-Protokolls gegenüber dem nativen Myrinetprotokoll entsteht.

	Netzwerk-schnittstelle	theoretische Leistung [MB/s]	Übertragung [MB/s]	Übertragung und Zuweisung [MB/s]
MPI 1	Myrinet (nativ)	250	233	233
RCF *(ByteBuffer)*	Myrinet (TCP)	250	220	180
RCF *(Vektorobjekte)*	Myrinet (TCP)	250	175	125
RCF *(ByteBuffer)*	Ethernet (TCP)	12,5	11,5	11
RCF *(Vektorobjekte)*	Ethernet (TCP)	12,5	10,5	10

Tabelle 17.2: *Performance für die Übertragung von Datenfeldern auf dem iRMB-PC-Cluster*

In Tab. 17.2 sind die durchschnittlichen Übertragungswerte des iRMB-PC-Clusters für Datenfelder mit unterschiedlichen Bibliotheken und Netzwerkschnittstellen dargestellt. Um beim Empfänger mit den Daten arbeiten zu können, müssen diese in VIRTUALFLUIDS u. U. den entsprechenden Zielfeldern zugewiesen werden. Dieser Mehraufwand wurde bei den Messwerten in der letzten Spalte berücksichtigt und entspricht der internen Umsetzung der mittels Multithreads umgesetzten RCF-Transmitter. Da bei MPI das Zielfeld direkt angeben werden kann, sind die Werte mit und ohne Zuweisung identisch. Aufgrund zusätzlicher interner Kopiervorgänge ist das Versenden der Vektorobjekte mit RCF zeitaufwändiger als die Verwendung von RCF-ByteBuffern. Letztere repräsentieren bei RCF den binären Datenstrom und weisen somit die beste Performance auf. Diese werden auch bei den RCF-Transmittern eingesetzt.

Ein weiteres Interprozessframework, das in diesem Zusammenhang getestet wurde, war das Multiscale Coupling Library and Environment (MUSCLE). Dieses versendet die Daten in sogenannten Conduits über ein auf Java [146] basierendes Agentenframework. Hierfür wurde ein spezieller MUSCLE-Transmitter auf Basis des Vektorpools entwickelt, dessen Performance, wie in Abb. 17.22a und 17.22b zu erkennen, etwas besser ist als die der RCF-Transmitter. Der Grund hierfür ist, dass die RCF-Transmitter die Daten blockierend senden und erst aus der Senderoutine zurückkehren, nachdem die Daten auf den anderen Prozess übertragen wurden. Dies könnte durch Verwendung von Einweg-Aufrufen weiter optimiert werden. Bei MUSCLE kann die nächste Methode gleich nach erfolgter Übergabe der Daten an das Agentenframework ausgeführt werden. Die Übertragung zum Zielprozess erfolgt im Hintergrund in einem separaten Java-Thread.

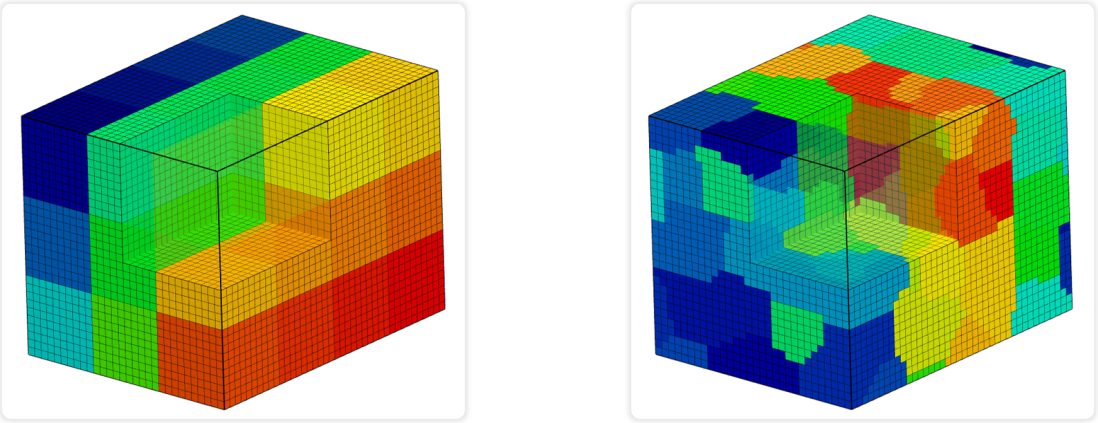
17.9 Skalierbarkeit und parallele Effizienz

Im Folgenden wird auf die Leistungsfähigkeit des parallelen Blockcodes eingegangen. Sämtliche Testrechnungen wurden mit dem iRMB-PC-Cluster (vgl. Kap. 11) auf Basis des inkompressiblen $d3q19$-BGK-Modells (vgl. Abschn. 3.1) durchgeführt. Der Quellcode wurde unter Ubuntu Linux Hardy (Version 8.04, 64-Bit) [160] mit der GNU Compiler Collection (GCC) Version 4.2.4 (64-Bit) [45] unter Verwendung bestmöglicher Optimierungsoptionen (u. a. O3) übersetzt. Für die MPI-Transmitter kam die MPI-1.2.7 MPICH-Portierung MPICH-GM der Firma Myricom zum Einsatz, die speziell auf die vorhandene Myrinet-Netzwerkkarten (2.000 MBit/s) angepasst wurde [107]. Die zusätzlich zur Verfügung stehende Ethernet-Schnittstelle wurde bei den Performancetests aufgrund der mit 100 MBit/s verhältnismäßig schlechten Bandbreite nicht verwendet. Wenn nicht anders angegeben, wurde bei Berechnungen mit bis zu einschließlich 40 Rechendiensten jeweils nur eine CPU pro Clusterknoten verwendet. Darüber hinaus kamen beide CPUs zum Einsatz. Hierbei wurde darauf geachtet, dass die Interprozesskommunikation innerhalb eines Clusterknotens weitestgehend vermieden wurde. Standardmäßig wurde mit doppelter Genauigkeit gerechnet. Simulationen, die mit einfacher Genauigkeit durchgeführt wurden, sind mit *SP* (*engl.: single precision*) gekennzeichnet.

17.9.1 Skalierbarkeit

17.9.1.1 Uniforme Blockgitter

Um den Scaleup zu ermitteln (vgl. Abschn. 11.2), wurde für die uniformen Testfälle die gleiche Konfiguration verwendet, wie bereits für den Transmittertest in Abs. 17.8.3. Dabei wurde sichergestellt, dass der CPU-eigene, lokale Arbeitsspeicher für das jeweilige Gitter ausreichend dimensioniert war und somit keine Zugriffe über Hypertransport auf den Speicher der zweiten CPU erfolgten (vgl. Abschn. 11.1). Neben der manuellen, homogenen Gebietszerlegung (vgl. Abb. 17.23a) wurde diese mit Hilfe des METIS-Moduls durchgeführt, wodurch man die für METIS typischen, geometrisch heterogenen Teilgebiete erhält (vgl. Abb. 17.23b).

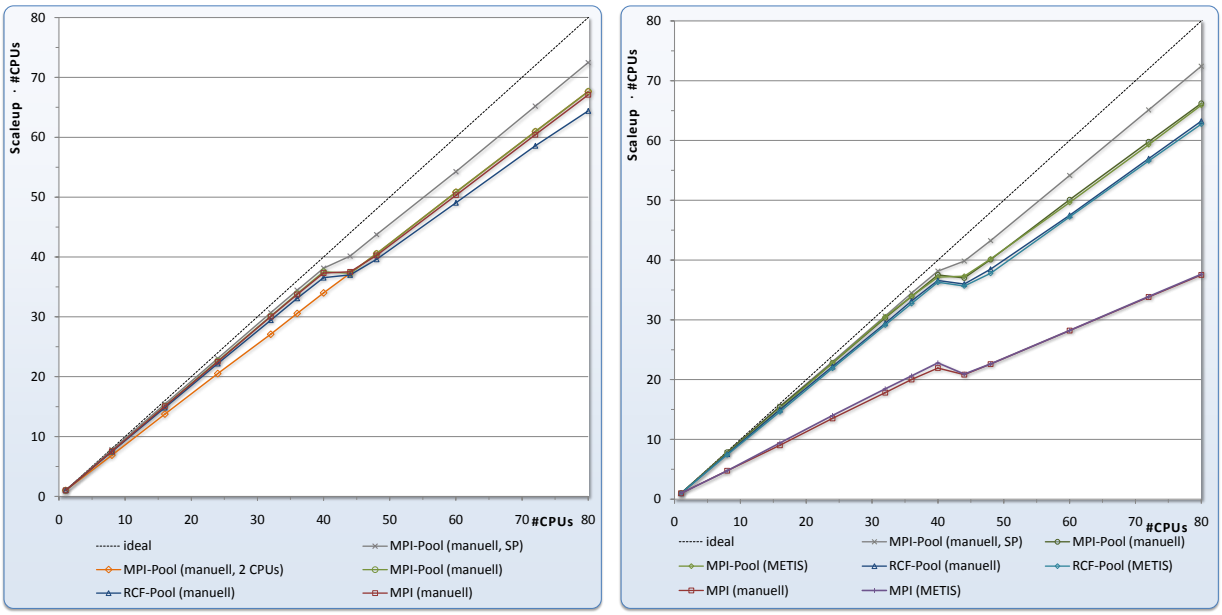


***(a)** Blöcke für 36 Rechenclients (manuell)* ***(b)** Blöcke für 36 Rechenclients (METIS)*

***Abbildung 17.23:** Uniforme Blockgitter für die Ermittlung des Scaleups*

Für den Testfall mit nur einem Block auf jedem Rechendienst, erhält man für die verschiedenen Transmittertypen vergleichbare Ergebnisse (Abb. 17.24a). Die Performance der RCF-Pooltransmitter

liegt hierbei etwas unterhalb der von den MPI-Pooltransmittern. Dies ist u. a. auf die Verwendung des TCP-Protokolls oder einer nachteiligen Unterstützung dessen seitens des Myrinet-Treibers zurückzuführen.



(a) *Ein Block mit je 100^3 Knoten pro Client* ***(b)*** *10^3 Blöcke mit je 10^3 Knoten pro Client*

Abbildung 17.24: *Scaleupergebnisse (uniforme Blockgitter)*

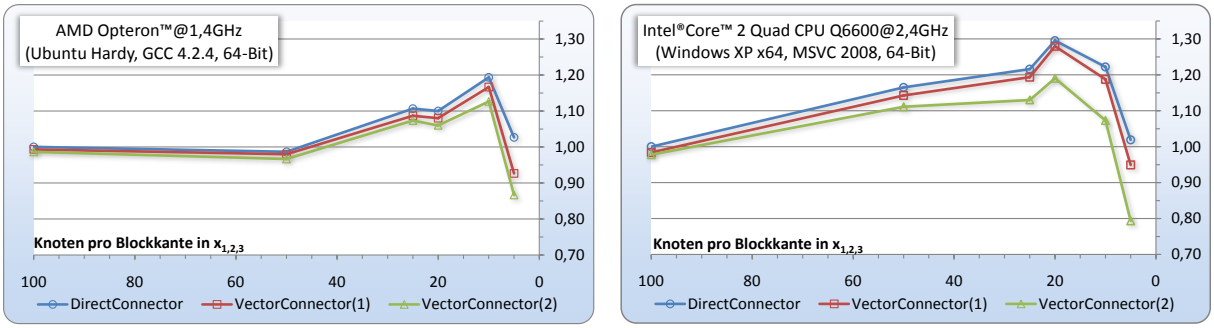
Auffällig ist der generelle Leistungseinbruch sobald beide CPUs eines Clusterknotens verwendet werden. Der Engpass ist hierbei die gemeinsam für die Kommunikation genutzte Myrinet-Netzwerkkarte, da bei gleichzeitigem Zugriff Bandbreitenprobleme die Folge sein können. Um dies zu verifizieren, wurde derselbe Testfall für den Bereich $2 \leq \#CPUs \leq 80$ mit zwei CPUs pro Mainboard unter Inanspruchnahme der leistungsstarken MPI-Pooltransmitter sowie einer manuellen Segmentierung gerechnet und ebenfalls mit dem Wert des seriellen Testfalls verglichen. Wie in Abb. 17.24a zu erkennen, entspricht das gemessen Leistungsdefizit für wenige CPUs im Verhältnis dem für Simulationen mit mehr als 40 Clients.

Als weiterer Test wurde das Problem unter Verwendung des Rechenkerns mit einfacher Genauigkeit gerechnet. Hierbei ist der Leistungsabfall für $\#CPUs > 40$ nicht so ausgeprägt wie bei Berechnungen mit doppelter Genauigkeit, wodurch die gleichzeitig über die gemeinsam genutzte Netzwerkkarte übertragene Datenmenge als Hauptverursacher bestätigt wird.

Erhöht man die Anzahl der Blöcke pro Rechendienst bei gleichzeitiger Reduzierung der Knotenanzahl pro Block, so weisen Berechnungen mit Einzelblockinterprozesskommunikation mit Standard-MPI-Transmittern die erwartete, bereits in Abs. 17.8.3 beschriebene, suboptimale Leistungsentwicklung auf (vgl. Abb. 17.24b). Diese ist auf die hohe Netzwerkaufrufanzahl und den damit verbundenen Latenzzeiten zurückzuführen. Im Gegensatz hierzu weisen die MPI- und RCF-Pooltransmitter durchweg gute Ergebnisse auf, wobei die über MPI kommunizierenden etwas effizienter sind. Die hier erstmalig verwendete METIS-Partitionierung lieferte trotz der geometrisch heterogenen Teilgebiete adäquate Resultate im Vergleich zu der manuellen Segmentierung.

Die serielle Geschwindigkeit belief sich beim ersten Testfall auf $1{,}40 \cdot 10^6$ nups (*SP:* $1{,}80 \cdot 10^6$) und beim zweiten trotz des erhöhten, lokalen Mehraufwandes auf $1{,}65 \cdot 10^6$ nups (*SP:* $1{,}95 \cdot 10^6$). Zurück-

zuführen ist diese etwa 20 %ige Leistungssteigerung auf die verbesserte Cachegängigkeit der kleineren Verteilungsmatrizen sowie auf die effiziente, lokale Interblockkommunikation. Um den Einfluss der Knotenmatrixgröße auf die Leistung zu ermitteln, wurde ein Gebiet in Form eines einzelnen Blocks mit 100^3 Knoten und periodischen Randbedingungen simuliert, der anschließend sukzessiv in kleinere Blöcke unterteilt wurde. Die Leistungsunterschiede im Vergleich zur Berechnung mit einem Block unter Verwendung von DirectConnectoren ist in Abb. 17.25 für verschiedene CPU- und Connectortypen dargestellt. Während bei VectorConnector(1) zwei Connectoren auf demselben Datenfeld operieren, verwenden sie bei VectorConnector(2) jeweils ein separates, sodass die Daten bei jeder Kommunikation zusätzlich zwischen diesen beiden Feldern kopiert werden müssen (vgl. Abschn. 15.2). In Abb. 17.25 ist zu erkennen, dass man bei VIRTUALFLUIDS mit kleineren Blöcken eine bessere Leitung erzielen kann als mit sehr großen. Bei sehr kleiner Knotenanzahl ($\leq 5^3$) kommt es aufgrund der erhöhten Interblockkommukation zu Leistungseinbußen. Die jeweils optimale Größe der zu verwendenden Verteilungsmatrix ist dabei abhängig von der jeweiligen Rechnerarchitektur, dem verwendeten Compiler und dem verwendeten LB-Modell. In Hinblick auf nicht-uniforme Blockgitter ist die effizientere Berechnung von VIRTUALFLUIDS mit kleineren Blöcken von Vorteil, da sich mit diesen geometrisch besser angepasste Gitter erstellen lassen (vgl. Abs. 16.3.1).



(a) *AMD-Opteron™@1,4 GHz* ***(b)*** *Intel® Core™ 2 Quad Q6600 CPU @2,4GHz*

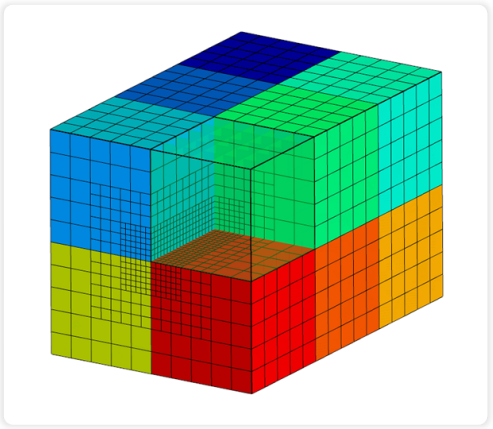
Abbildung 17.25: *Einfluss der Knotenmatrixgröße auf die Gesamtleistung*

Der in Abschn. 10.6 beschriebene signifikant hohe Speicherbedarf von METIS bei der Gebietszerlegung ist bei dem Blockgitteransatz infolge der Beschränkung auf die Blöcke nicht gegeben. Für den Testfall mit 80 Rechenprozessen und 1.000 Blöcken pro Prozess wurde die Segmentierung mit nur 80.000 Blöcken anstatt mit den vorhandenen $80 \cdot 10^6$ Rechenknoten durchgeführt. Auch der Speicherbedarf des TopologyServices ist für diesen Fall mit 350 MB akzeptabel und entsprach ungefähr dem eines einzelnen Rechendienstes, der $1 \cdot 10^6$ Berechnungsknoten vorhielt. Alternativ kann hier ein anderer TopologyService-Typ verwendet werden, der im Gegensatz zu dem benutzten, über keinerlei Knoteninformationen verfügt. Dadurch wird der durch diesen Dienst beanspruchte Speicher auf ein Minimum reduziert. Durch die fehlende Fluidknoteninformation kann es aber zu Nachteilen bei der Blockwichtung kommen.

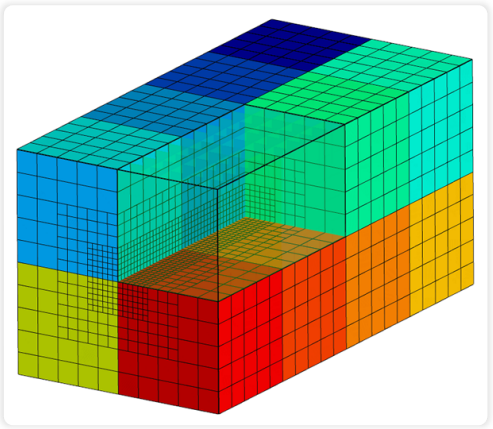
17.9.1.2 Nicht-uniforme Blockgitter

Zur Bestimmung des Scaleups für Simulationen mit nicht-uniformen Gittern kamen die in Abb. 17.26 dargestellten Blockgitter, die in alle Raumrichtungen mit periodischen Randbedingungen versehen wurden und jeweils 10^3 Berechnungsknoten je Block aufwiesen, zum Einsatz. Die Verfeinerung wurde dabei so gewählt, dass bei Erhöhung der CPU-Zahl die einzelnen Teilgebiete identisch blieben.

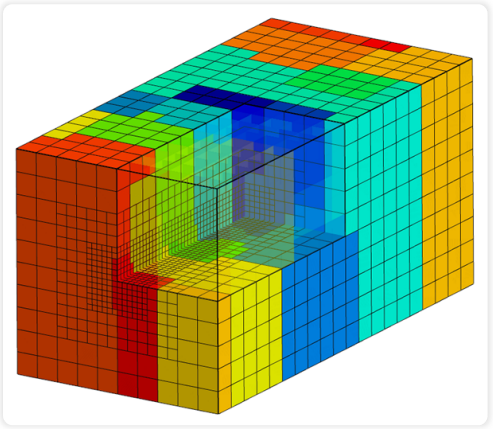
Bei der Verwendung von METIS wurde die Segmentierung zum einen gemäß Abschn. 17.4 für einen Graphen des Gesamtblockgitters durchgeführt (METIS standard, Abb. 17.26c) und zum anderen für jeden Level separat für alle vorhandenen Rechenclients vorgenommen, wobei hierbei aufgrund einer nicht trivialen Implementierung die Konnektivität zwischen Blöcken unterschiedlicher Level nicht berücksichtigt wurde (METIS levelweise, Abb. 17.26d).



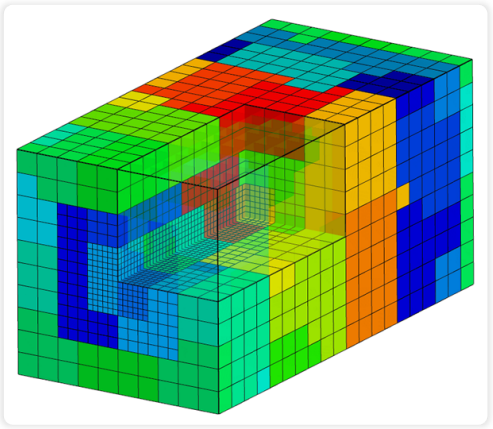
(a) *Blöcke für 12 Rechenclients (manuell)*



(b) *Blöcke für 16 Rechenclients (manuell)*



(c) *Blöcke für 16 Clients (METIS standard)*



(d) *Blöcke für 16 Rechenclients (METIS levelweise)*

Abbildung 17.26: *Nicht-uniforme Blockgitter für die Ermittlung des Scaleups*

Infolge der Implementierung der Algorithmen zur Skalierung der numerischen Daten zwischen den einzelnen Gitterleveln, müssen in der aktuellen Version von VIRTUALFLUIDS im feinstem Zeitschritt bis zu dreimal Daten zwischen den einzelnen Prozessen ausgetauscht werden anstatt einmal, wie es Berechnungen uniformen Gittern der Fall ist. Dadurch kommt es u. a. zu einem zeitlichen Mehraufwand, der sich durch die dreifache Initialisierung des Datentransfers ergibt. Dennoch ist der gemessene Scaleup für die manuelle Gebietszerlegung bei Nutzung der MPI-Pooltransmitter, u. a. aufgrund der geringen Latenzzeit des Myrinet-Netzwerkes und der Optimierung der Datenübertragung (vgl. Abs. 17.8.2), gleichwertig zu dem der uniformen Blockgitter (Abb. 17.27). Für die serielle Simulation wurde hier eine Leistung von $1,55 \cdot 10^6$ nups erzielt. Bei den RCF-Pooltransmittern macht sich die erhöhte Kommunikationsanzahl jedoch negativ bemerkbar sobald beide CPUs eines Clusterknotens verwendet werden. Gründe hierfür können, wie erwähnt, die TCP-Kommunikation über das Myrinet-Netzwerk als auch die Art der Transmitterimplementierung sein.

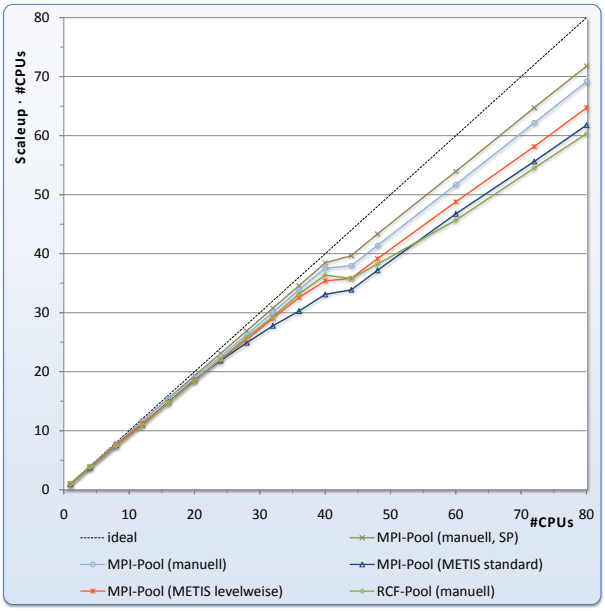


Abbildung 17.27: *Scaleupergebnisse (nicht-uniforme Blockgitter)*

Auffallend ist, dass die Standard-METIS-Segmentierung in einer schlechteren Performance resultiert als die Verwendung der levelweisen Segmentierung. Ursache hierfür ist, dass eine Partitionierung mit Hilfe des Gesamtgraphen nur einen einheitlichen Arbeitsaufwand während des gröbsten Zeitschrittes, nicht aber eine gleichförmige Verteilung dessen zwischen den einzelnen Datenaustauschpunkten gewährleistet. Dadurch kann es zu Wartezeit auf den einzelnen Clients kommen, die sich letztlich negativ auf die Gesamtleistung auswirkt. Im Gegensatz zu den uniformen Blockgittern resultiert hier die Verwendung der geometrisch heterogenen Teilgitter der METIS-Zerlegung aufgrund des damit verbundenen unterschiedlichen Interprozesskommunikationsaufwandes und den drei Synchronisierungspunkten in einer schlechteren Skalierung.

17.9.2 Parallele Effizienz

17.9.2.1 Uniforme Blockgitter

Die Gitterkonfiguration für die Ermittlung der parallelen Effizienz (vgl. Abschn. 11.1) ist in Abb. 17.28 dargestellt. Dabei handelt es sich um eine Kugel in einem Rechteckkanal, der seitlich durch Wände mit no-slip-Randbedingungen abgegrenzt wurde. Am Einlass befindet sich eine Einstrom- und am Auslass eine Druckrandbedingung. Das Gebiet wurde mit unterschiedlicher Blockanzahl mit je 11^3 Knoten je Block diskretisiert, um die parallele Effizienz für verschiedene Gitterknotenauflösungen ermitteln zu können. Der Testfall mit $1,4 \cdot 10^7$ Knoten benötigte bei der Berechnung mit einem Rechendienst 85 % des Gesamtspeichers eines Clusterknotens und musste somit über Hypertransport auf den Speicher der zweiten CPU zugreifen (vgl. Abschn. 11.1). Die erzielte Leistung lag bei $1,50 \cdot 10^6$ nups (*SP:* $1,85 \cdot 10^6$ nups bei 43 % Speicherauslastung) während der Testfall mit $7 \cdot 10^6$ Knoten und einer knapp 45 %igen Arbeitsspeicherbelegung $1,57 \cdot 10^6$ nups erreichte. Die Gebietszerlegung erfolgte hier ausschließlich unter Verwendung der entsprechenden METIS-Module.

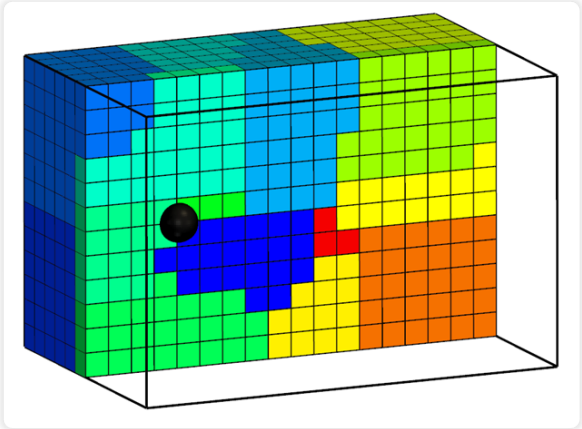


Abbildung 17.28: *Uniforme Blockgitter zur Ermittlung der parallelen Effizienz*

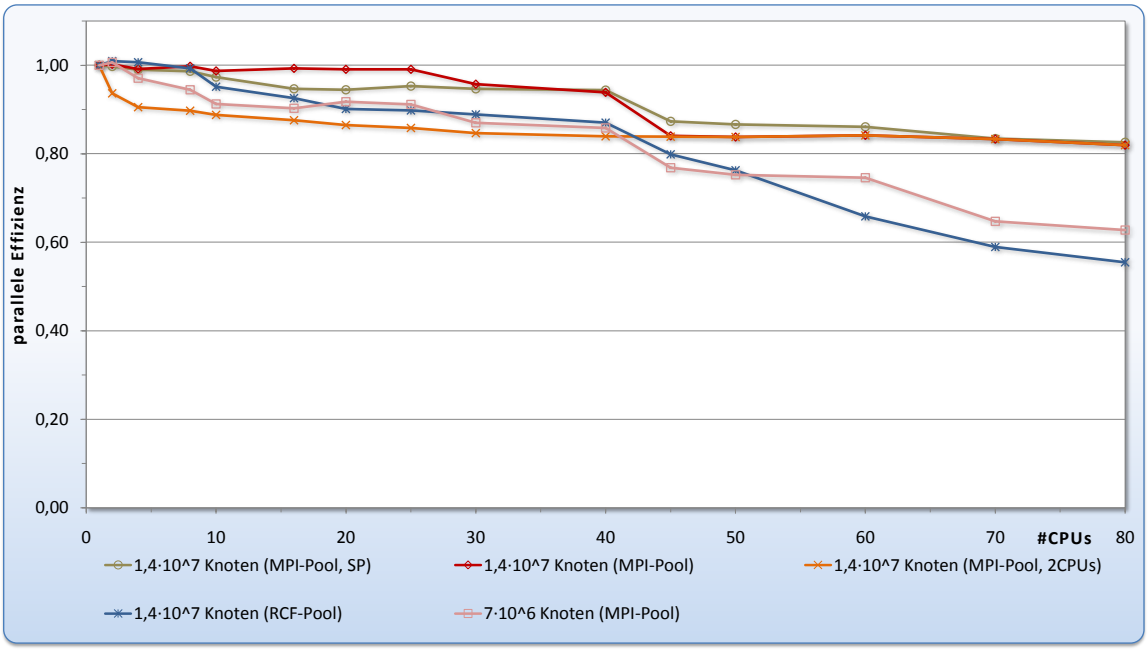


Abbildung 17.29: *Parallele Effizienz (uniforme Blockgitter)*

Vergleicht man die Ergebnisse der MPI-Pooltransmitter für verschiedene Gitterauflösungen, so fällt die Performance, wie schon beim Scaleup, stark ab sobald zwei CPUs auf einem Clusterknoten verwendet werden. Auch hier ist dafür der erhöhte, gleichzeitige Datentransfer der beiden Rechenprozesse über eine Netzwerkschnittstelle verantwortlich.

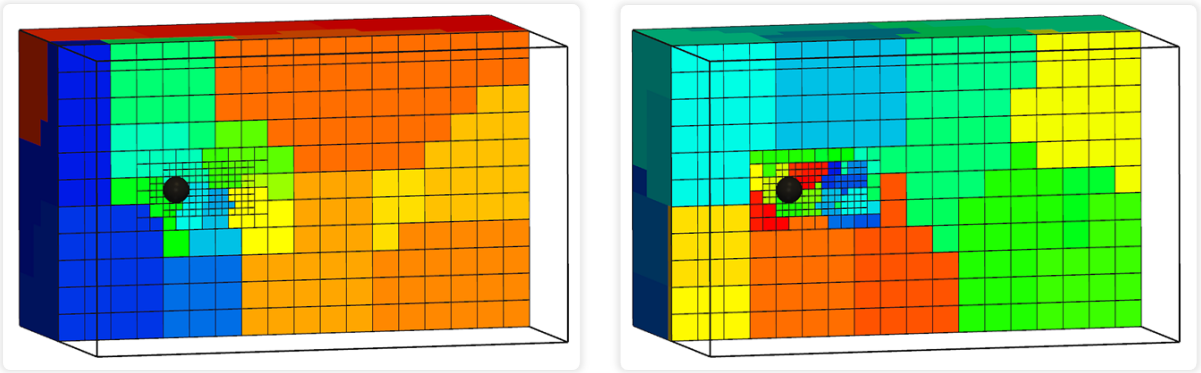
Beim Testfall mit $1,4 \cdot 10^7$ Knoten kommt es für eine hohe Anzahl an Rechendiensten erwartungsgemäß zu einem signifikanten Einbruch der parallelen Effizienz. Bei 80 aktiven Rechenclients befinden sich auf jedem Rechenprozess nur noch knapp 90.000 Rechenknoten, sodass sich die erhöhte Frequenz des Datenaustausches und die Reduzierung des lokalen Rechenaufwandes negativ auf die Gesamtleistung auswirkt (Abb. 17.29). Zudem ist zu beachten, dass die Gebietszerlegung ausschließlich für Blöcke durchgeführt wird. Ist aufgrund der vorhandenen Blockanzahl eine homogene Verteilung

nicht möglich, so ist infolge der damit verbundenen nachteiligen Lastverteilung eine verminderte Performance die Folge.

Bei den RCF-Pooltransmittern wird der bereits bei der Messung des Scaleups erkennbare Leistungsverlust bei hoher CPU-Anzahl bestätigt. Die MPI-Pooltransmitter hingegen weisen, solange die Gebietsgröße auf den einzelnen Rechenprozessen ausreichend groß ist, eine durchweg gute Leistung auf.

17.9.2.2 Nicht-uniforme Blockgitter

Als Testgebiet wurde auch hier eine Rechteckkanal mit einer umströmten Kugel und einer Gitterauflösung von $1,4 \cdot 10^7$ Knoten verwendet. Im Gegensatz zu Abs. 17.9.2.1 wurde das Gitter hierbei im Bereich der Kugel verfeinert und für die Blockverteilung, wie bereits in Abs. 17.9.1.2, zwei verschiedene METIS-Segmentierungsmodule verwendet (Abb. 17.30).



(a) *METIS standard* ***(b)*** *METIS levelweise*

Abbildung 17.30: *Nicht-uniforme Blockgitter zur Ermittlung der parallelen Effizienz*

Die parallele Effizienz bei Simulationen deren Blockaufteilung auf der Graphenzerlegung für das Gesamtgitter beruht, reduziert sich bereits für wenige Rechenprozesse auf knapp 65 % und der Unterschied zu Berechnungen mit zwei CPUs pro Knoten ist vernachlässigbar (Abb. 17.31). Demgegenüber liegt die parallele Effizienz der levelweisen Segmentierung durchschnittlich 35 % darüber und erreicht bei 80 CPUs immer noch 78 %. Wie in Abs. 17.9.1.2 beschrieben, ist hier die Arbeit innerhalb der gestaffelten Zeitschleife insgesamt gleichmäßiger verteilt. Eine noch höhere Leistung könnte erzielt werden, wenn die numerische Skalierung zwischen Blöcken unterschiedlicher Gitterauflösung, sofern möglich, lokal durchgeführt werden würde, da bei der Skalierung eine größere Datenmenge übertragen wird als bei den SameLevelConnectoren (vgl. Abschn. 15.2). Wie in Abb. 17.30b zu erkennen, ist diese Eigenschaft derzeit noch nicht gegeben. Die dreifache Kommunikation pro Nachbarprozess und feinstem Zeitschritt wirkt sich hier insbesondere bei einer Nutzung von 44 oder mehr CPUs bei den RCF-Pooltransmittern sehr negativ auf die Effizienz aus. Selbst die Verwendung der levelweisen Blockunterteilung mit METIS liefert nur eine geringfügige Verbesserung.

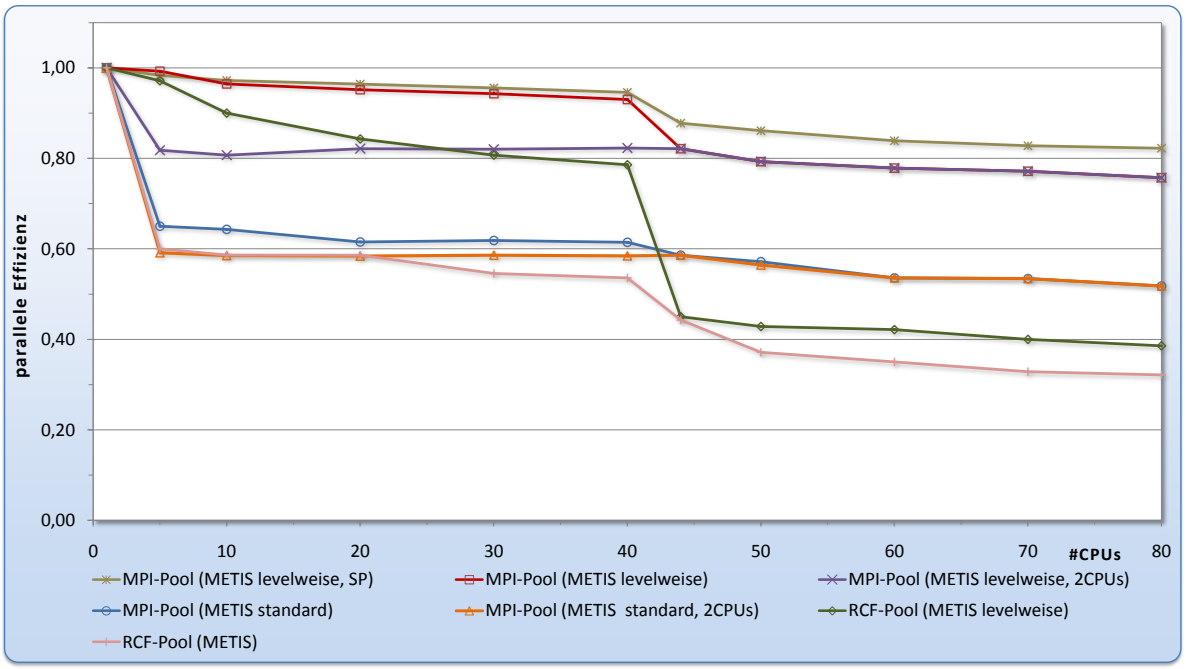


Abbildung 17.31: *Parallele Effizienz (nicht-uniforme Blockgitter)*

17.9.3 Zusammenfassung

Insgesamt ist die erzielte Leistung des auf Blockgitter basierenden VIRTUALFLUIDS zufriedenstellend. Insbesondere die MPI-Pooltransmitter wiesen eine durchweg gute Performance auf und sind stets den Einzeltransmittern vorzuziehen. Die Verwendung von Verteilungsmatrizen mit geringen Abmessungen resultiert aufgrund von Sekundäreffekten, wie einer verbesserten Zugriffszeit auf den Cache, im Allgemeinen in besseren Ergebnissen. Bei RCF-Pooltransmittern hingegen traten hohe Performanceeinbußen bei erhöhter CPU-Zahl auf. Hier ist u. a. zu untersuchen, ob dieser Effekt auf die Kombination TCP-Protokoll und Myrinet-Netzwerkkarten-Treiber zurückzuführen ist oder ob diesbezüglich die Leistung durch Änderungen in der Implementierung verbessert werden kann.

Generell muss bei der Gebietszerlegung nicht-uniformer Gitter verstärkt darauf geachtet werden, dass der Speicherbedarf der resultierenden Teilgitter nicht den zur Verfügung stehenden Arbeitsspeicher des jeweiligen Zielrechners überschreitet. Aufgrund der unterschiedlichen Wichtung kann es bei der Partitionierung nicht-uniformer Gitter vorkommen, dass einem Client wenige Blöcke mit hohen und einem anderen viele Blöcke mit niedrigen Wichtungen zugewiesen werden und somit der zur Verfügung stehende Speicher nicht ausreicht. Hierauf hat man bei der METIS-Segmentierung jedoch nur bedingt Einfluss.

Alternative Optimierungen sind durchaus denkbar. So könnte in weiteren Entwicklungsstufen für Simulationen mit *aktiver* Netzwerkhardware eine asynchrone Berechnung innerhalb der Teilgitter realisiert werden, bei der die LB-Algorithmen zunächst für Knoten an den Prozessrändern durchgeführt werden, sodass die Daten nicht blockierend versendet werden während das Gebietsinnere berechnet wird. Bei nicht-uniformen Gittern ist die Reduktion des dreifachen Datenaustausches auf einen pro feinem Zeitschritt sowie eine weitere Optimierung der Gebietszerlegung notwendig. Aufgrund der beschränkten Einflussnahme auf die METIS-Partitionierung sollten weitere Zerleger auf deren Anwendbarkeit hin untersucht werden.

17.10 Anwendungsbeispiele

In diesem Abschnitt werden einige Anwendungsbeispiele gezeigt, die mit der verteilt rechnenden, blockgitterbasierten Erweiterung von VIRTUALFLUIDS simuliert wurden. Für weitere vorwiegend mit der seriellen Version des Blockgittercodes durchgeführte Simulationen wird auf [46] verwiesen. Neben der Validierung des Strömungskerns anhand einer umströmten Kugel bei $Re = 10.000$, der Verifikation der verteilten Fluid-Struktur-Interaktion und ein Beispiel für die parallele Gitteradaptivität werden anhand der Schallabsorption poröser Medien und einer Kühlturmumströmung zwei praktische Beispiele aus dem Ingenieursbereich präsentiert.

17.10.1 Turbulente Strömung um eine Kugel

In diesem Abschnitt wird die turbulente Umströmung einer Kugel für eine Reynoldszahl von 10.000 simuliert (Abb. 17.32).

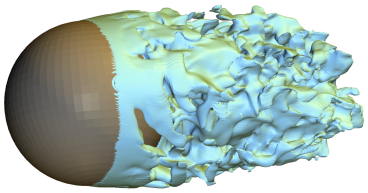


Abbildung 17.32: *Isokontur von* $u_{x_1} = 0$

Die Berechnung von turbulenten Strömungen gehört zu den anspruchsvollsten Aufgaben in der Strömungsmechanik, da bei diesen Phänomene räumlich und zeitlich auf sehr unterschiedlichen Skalen stattfinden. Bei einer direkten numerischen Simulation (DNS) versucht man diese entsprechend in Raum und Zeit aufzulösen. Aufgrund des damit verbundenen hohen Rechenaufwandes sind derartige Berechnungen derzeit auf verhältnismäßig kleine Gebiete und verhältnismäßig niedrige Reynoldszahlen beschränkt. Eine Alternative zu diesen ist die Verwendung von Turbulenzmodellen, bei denen kleine Skalen modelliert und höhere aufgelöst werden.

In diesem Beispiel kommt das Smagorinsky-Large-Eddy-Turbulenzmodell zum Einsatz [120]. Dabei werden die kleinen nicht aufgelösten Wirbel über die Einführung einer turbulenten Viskosität ν_T modelliert. Im LB-Kontext wird diese mit der Smagorinsky-Konstante C_S und dem Dehnungstensor ϵ folgendermaßen ermittelt [140]:

$$\nu_T = (C_S \, \Delta x_l)^2 \, |\epsilon| \qquad \text{mit} \qquad \epsilon_{\alpha\beta} = \frac{1}{2}\left(\frac{\partial u_\alpha}{\partial x_\beta} + \frac{\partial u_\beta}{\partial x_\alpha}\right) \tag{17.2}$$

Die lokale Relaxationszeit τ_{total} (vgl. Gl. 3.5) ergibt sich unter Berücksichtigung der molekularen Viskosität ν_0 zu [94]:

$$\tau_{total} = \frac{3}{c^2}\nu_{total} + \frac{1}{2}\Delta t_l = \frac{3}{c^2}(\nu_0 + \nu_T) + \frac{1}{2}\Delta t_l \tag{17.3}$$

Unter Berücksichtigung des Spannungstensors lässt sich folgende Beziehung herleiten:

$$\tau_{total} = \tau_0 + \frac{\sqrt{\tau_0^2 + \frac{18\, C_S^2\, \Delta t_l{}^2\, Q}{c^2}} - \tau_0}{2} \tag{17.4}$$

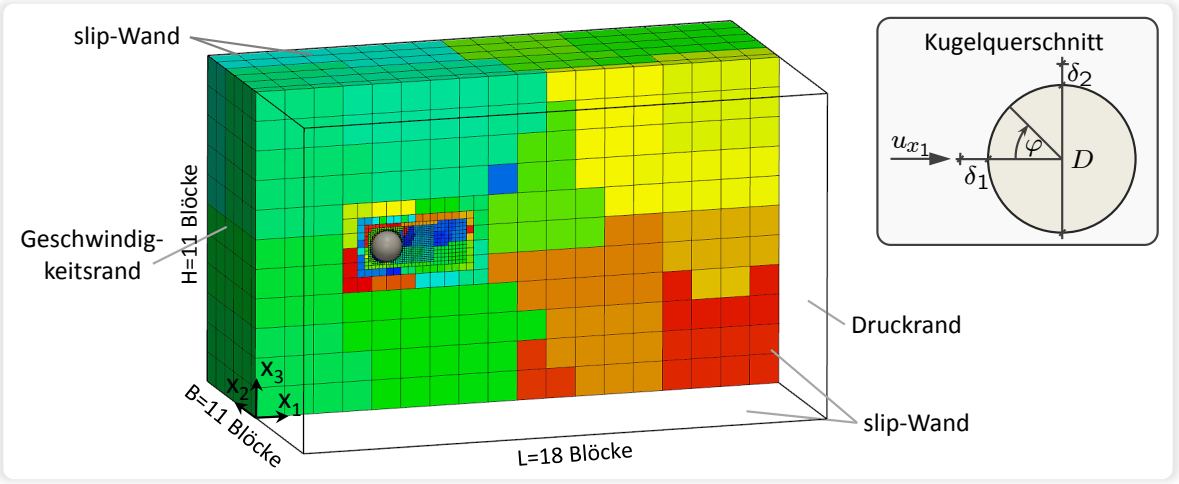
mit

$$Q = \sqrt{\sum_{\alpha\beta} 2\,\Pi_{\alpha\beta}\,\Pi_{\alpha\beta}}$$

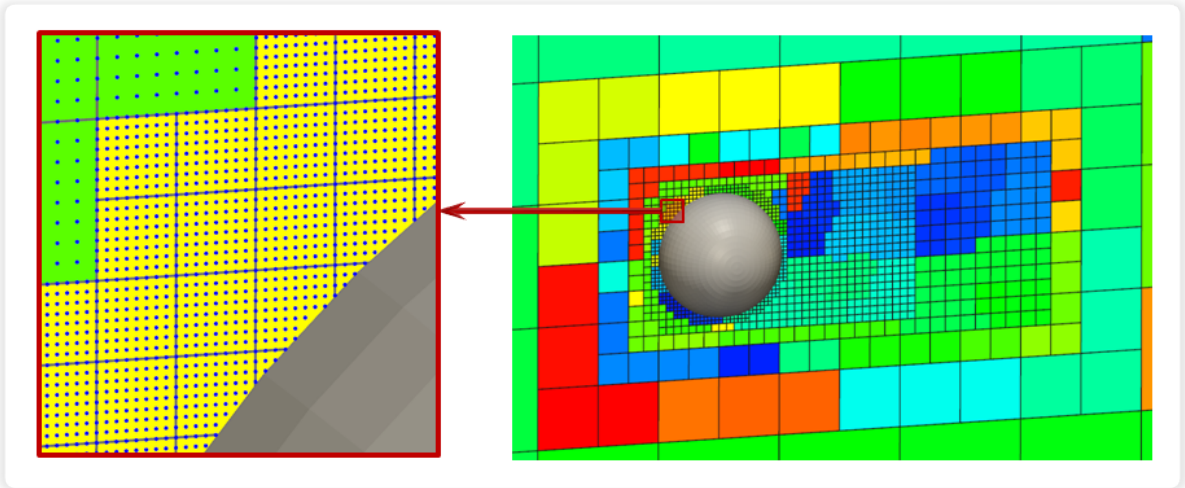
wobei für $\Pi_{\alpha\beta}$ Gl. 3.14 Anwendung findet.

Weiterführende Informationen zu LB-Simulationen mit Turbulenzmodellen sind in [79, 92, 94, 167] zu finden.

Die hier durchgeführte Simulation erfolgte mit dem 3-D-Einphasenkern von VIRTUALFLUIDS unter Verwendung des Momentenmodells und der eben beschriebenen Turbulenzerweiterung. Die notwendige Skalierung der Informationen zwischen den einzelnen Gitterleveln wurde dabei unverändert mit der in Abschn. 3.3 beschriebenen Methode für das MRT-Modell realisiert.



(a) *Blocksetup für umströmte Kugel (farbig: levelweise METIS Zerlegung)*



(b) *Vergrößerter Gebietsausschnitt im Bereich der Kugeloberfläche*

Abbildung 17.33: *Gitterkonfiguration für eine Kugelumströmung bei* $Re = 10.000$

Die mit der die Boundary-Fitting-Methode berücksichtigte Kugel befand sich in einem Rechteckkanal, dessen Wände mit der slip-Randbedingung diskretisiert wurden und dessen Kanalbreite dem elffachen Kugeldurchmesser D entsprach (Abb. 17.33). Insgesamt umfasste das Gebiet 28.393 Blöcke

mit jeweils 9^3 Knoten ($\approx 2 \cdot 10^7$ Knoten). In einem groben Zeitschritt erfolgten $\approx 4 \cdot 10^8$ Knoteniterationen (Tab. 17.3). Eine Simulation mit einem uniformen Gitter und derselben Kugelauflösung von 256 Knoten über den Durchmesser würde eine um zwei Größenordnungen höhere Anzahl an Berechnungsknoten ($\approx 5 \cdot 10^{10}$) benötigen.

Level	#Fluidblöcke	#Fluidknoten	Knotenwichtung	gewichtete Summe
0	2.133	1.554.957	1	1.554.957
1	232	169.128	2	338.256
2	524	381.996	4	1.527.984
3	2.312	1.685.448	8	13.483.584
4	10.008	7.295.832	16	116.733.312
5	13.184	7.888.904	32	252.444.928
Summe	28.393	18.976.265	-	386.083.021

Tabelle 17.3: *Gebietsstatistik*

Während der Berechnung wurde die vom Fluid auf die Kugel ausgehende Kraft kontinuierlich gemessen und der Kraftbeiwert analog zur Kugelumströmung in Abschn. 7.4 mit Gl. 7.14 ermittelt. Der resultierende Verlauf ist in Abb. 17.34 dargestellt.

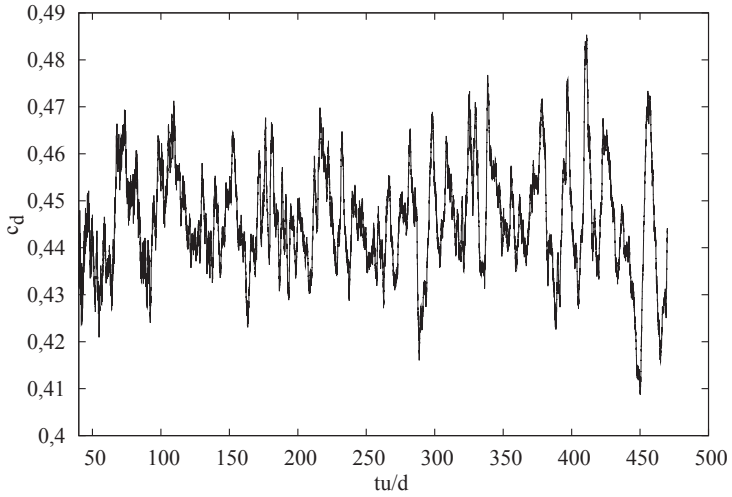
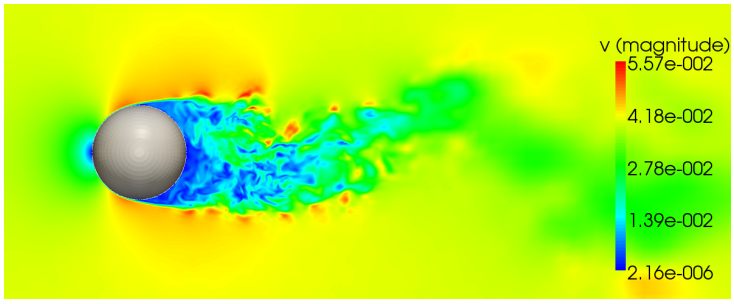


Abbildung 17.34: *Zeitlicher Verlauf des Kraftbeiwertes c_d für Re = 10.000*

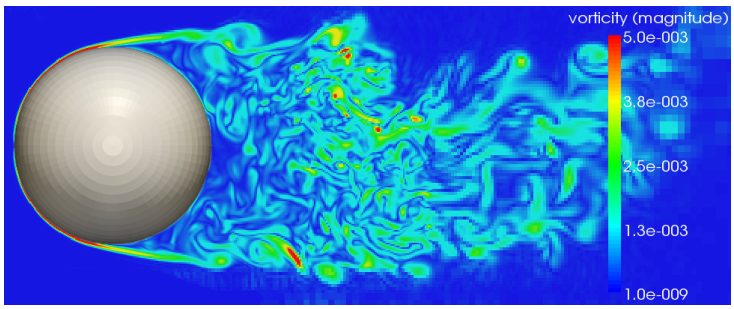
Als Maß für den lokalen Druck ist der Druckbeiwert c_p (Gl. 17.5) eine weitere wichtige Kennzahl.

$$c_p = \frac{p - p_\infty}{\frac{1}{2}\,\rho_\infty\,u_\infty^2} \tag{17.5}$$

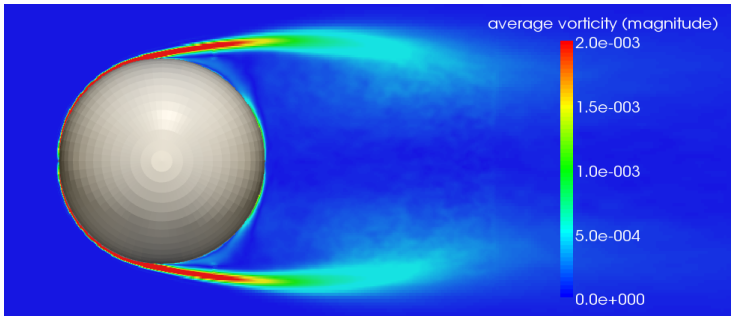
Hierbei ist p_∞ der ungestörte Druck bei einer ungestörten Anströmgeschwindigkeit u_∞ des umgebenden Fluides mit einer Dichte von ρ_∞ und p der an einem Punkt der Kugeloberfläche gemessene Druck. Der entlang der Kugeloberfläche zeitlich über $1 \cdot 10^5$ grobe bzw. $32 \cdot 10^5$ feine Berechnungsschritte gemittelte Druckbeiwert stimmt sehr gut mit den in der Literatur gefunden Vergleichswerten überein (Abb. 17.36).



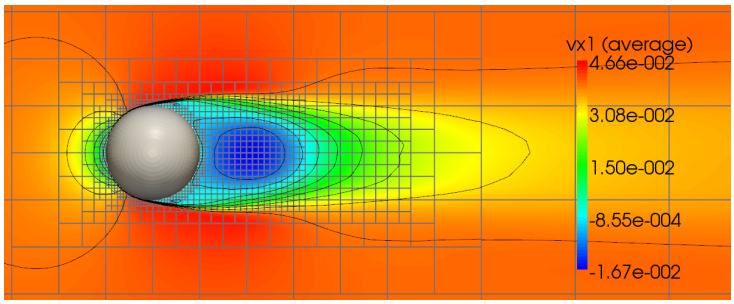
(a) *Geschwindigkeitsbetrag*



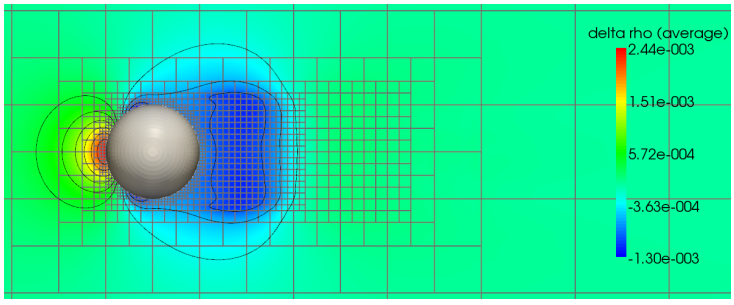
(b) *Wirbelstärke*



(c) *Gemittelte Wirbelstärke*

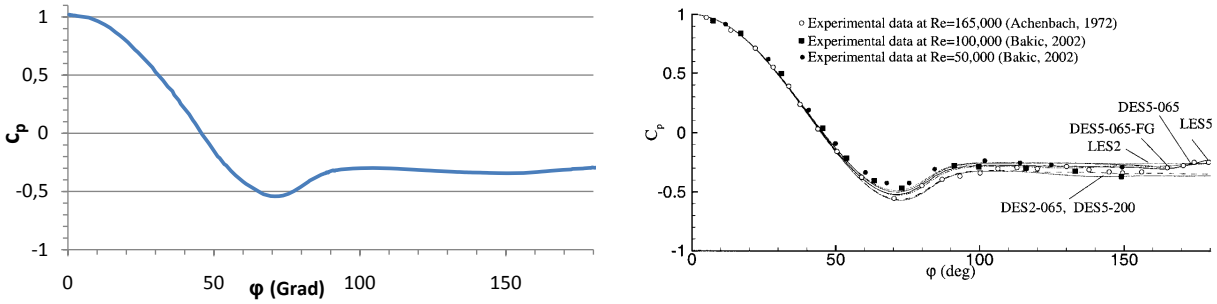


(d) *Gemittelte Geschwindigkeitskomponente* u_{x_1}



(e) *Gemittelte Dichtevariation*

Abbildung 17.35: *Simulationsergebnisse einer umströmten Kugel bei* $Re = 10.000$ *(Schnitt durch Gebietsmitte)*



(a) VIRTUALFLUIDS *mit LES* ***(b)*** *Referenzwerte aus [21]*

Abbildung 17.36: *Gemittelter Druckbeiwert entlang der Kugeloberfläche (Mittelebene)*

In Abb. 17.35 sind weitere Simulationsergebnisse dargestellt. Die Durchschnittswerte wurden dabei über $1 \cdot 10^5$ grobe Zeitschritte hinweg gemittelt. Die Wirbelstärke lässt erkennen, wie komplex die Strömung hinter der Kugel ist. Die sich aus $\delta_1 = 1{,}3\ D/\sqrt{Re}$ und $\delta_2 = 1{,}06\ D\ /\ \sqrt{Re}$ am Staupunkt der Kugel [19] ergebende Grenzschichtdicke entspricht in diesem Beispiel ungefähr drei Knoten und stimmt mit der aus den gemittelten Ergebnissen gewonnenen Grenzschichtdicke qualitativ überein.

Die erzielte parallele Effizienz für diese Simulation mit 70 Rechendiensten, bei der jeweils beide CPUs der iRMB-PC-Clusterknoten verwendet wurden, lag bei 80 % und somit etwas oberhalb des in Abs. 17.9.2.2 erzielten Wertes. Ein Grund hierfür ist u. a. die geringere serielle Leistung des Rechenkerns bei Verwendung des Momentenmodells mit der LES-Turbulenzerweiterung ($\approx 9{,}5 \cdot 10^5$ nups), sodass der Interprozesskommunikationsaufwand weniger Einfluss auf die parallele Leistungsfähigkeit hat.

17.10.2 Fallende Kugel

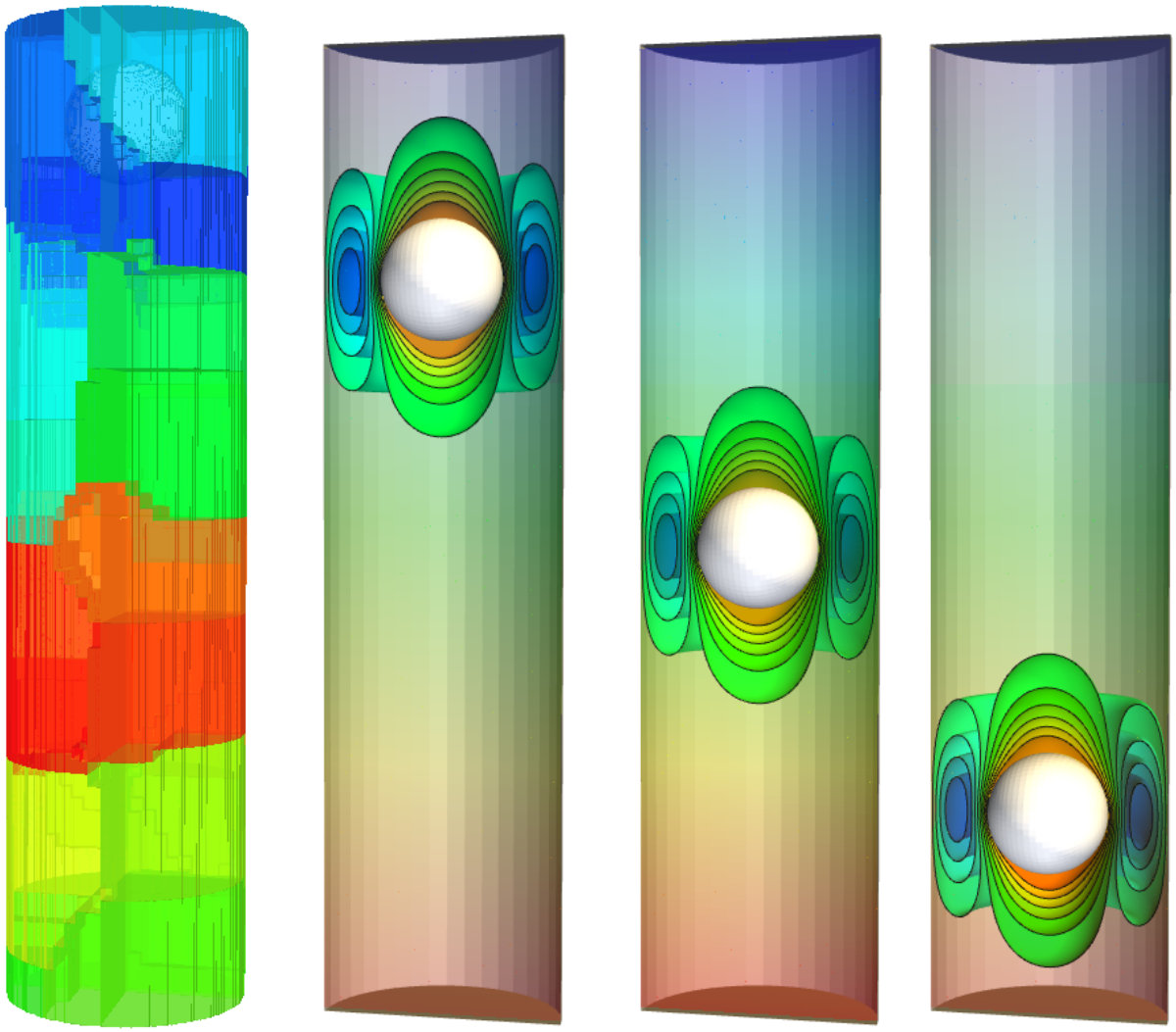
Zur Validierung der verteilten Fluid-Struktur-Interaktion (vgl. Abs. 17.7.4) wurde eine fallende Kugel in einem Rohr mit ruhender Flüssigkeit simuliert. Der Zylinderdurchmesser $D_{Zylinder}$ entspricht hierbei dem doppelten Kugeldurchmesser D_{Kugel} und das Dichteverhältnis vom Kugelmaterial zum umgebenden Fluid beträgt 2 : 1. Als Strukturlöser wurde die PHYSICSENGINE von Klaus Iglberger [81] verwendet. Die Berechnungen mit dem in dieser Arbeit erstellten parallelem Löser wurden von Sebastian Geller durchgeführt. Für eine genaue Beschreibung der Berechnungsparameter, wie z. B. die iterative Ermittlung der analytischen Fallgeschwindigkeit, und weitere Ergebnisse wird auf [46] verwiesen.

Erste Simulationen für verschiedene Gitterauflösungen lieferten mit einem relativen Fehler von ungefähr einem Prozent eine gute Übereinstimmung mit der analytischen Lösung (Tab. 17.4).

Gebietsgröße ($D^2_{Zylinder} \times H$)	D_{Kugel}	u_{analyt}	u_{num}	rel. Fehler [%]
$120^2 \times 480$	60	0,0906	0,09190	1,415
$150^2 \times 600$	75	0,0906	0,09180	1,307
$180^2 \times 720$	90	0,0906	0,09177	1,275
$210^2 \times 840$	105	0,0906	0,09174	1,243

Tabelle 17.4: *Ergebnisse für Testfall „Fallende Kugel“*

In Abb. 17.37 ist exemplarisch für den Testfall mit einer Gitterauflösung von $150^2 \times 600$ die Gebietszerlegung mit METIS für das uniforme Gitter sowie die Isokonturen der Geschwindigkeitskomponente u_{x_3} und als Schnitt die vorhandene Druckverteilung des Fluides in der Zylindermitte für verschiedene Zeitpunkte dargestellt.



(a) *Teilgebiete (METIS)* ***(b)*** $t = 2s$ ***(c)*** $t = 4,5s$ ***(d)*** $t = 7s$

Abbildung 17.37: *Simulation einer fallenden Kugel*

Die parallele Effizienz lag bei 20 Rechendiensten im Schnitt bei 85 %. Leistungsmindernd war hier die nicht optimale Lastverteilung durch die ausschließlich a priori durchgeführte Gebietszerlegung. Die Verwendung des zentralen Dienstes, der den Fluid- und den Strukturlöser miteinander koppelt, kann bei einer Vielzahl zu berücksichtigender Festkörperobjekte zeitkritisch werden. Daher ist für verteilte Simulationen in solchen Fällen eine prozessweise Kopplung der Löser wünschenswert. Hierfür muss der Strukturlöser ein entsprechendes Interface zur Verfügung stellen. Im Gegensatz zur Adhoc [32] bietet die parallele Version der PHYSICSENGINE ein solches an. Dadurch können die Teilgitter von VIRTUALFLUIDS direkt mit den Teilprozessen des Strukturlösers kommunizieren und der Informationsaustausch über den zentralen InteractorService ist nicht mehr erforderlich. Hierfür müssen entsprechende SteeringPlugin-Module erstellt werden, die direkt mit den jeweiligen Strukturkörpern der

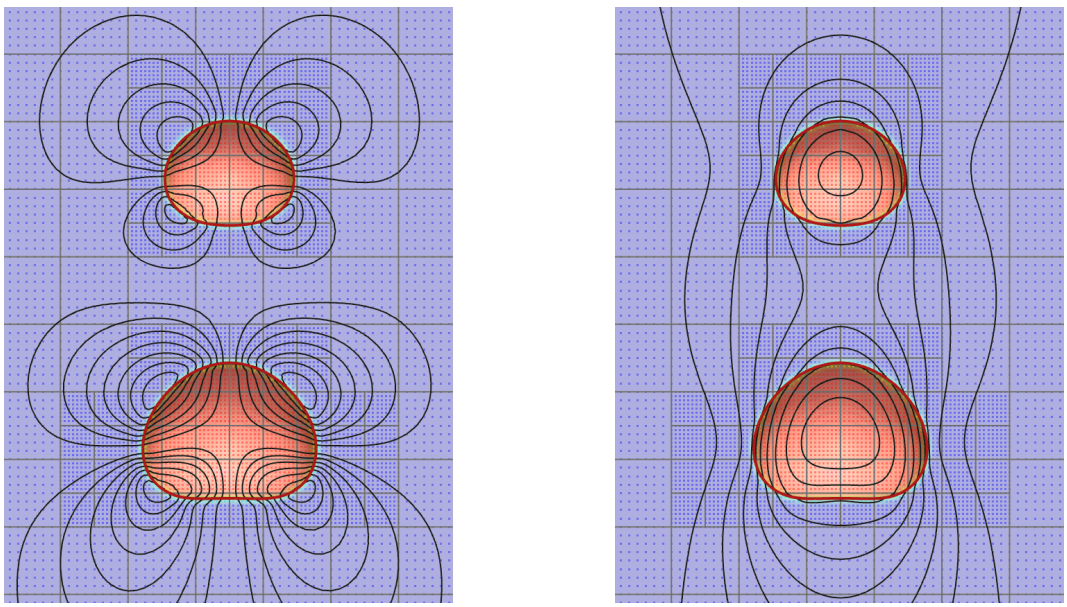
PHYSICSENGINE (pe::bodies) Daten austauschen. Eine solche Kopplung befindet sich derzeit in der Entwicklungsphase.

17.10.3 Aufsteigende Blasen

Zu Validierung der parallelen Gitteradaptivität wurden zwei aufsteigende Blasen unterschiedlicher Größe in einem Zylinder simuliert. Durch das größere Volumen der unteren Blase hat diese eine größere Auftriebskraft als die kleinere, voran laufende. Zudem erfährt die voluminösere Blase durch den Sog im Nachlaufgebiet der oberen einen geringeren Auftriebswiderstand, sodass die beiden Blasen letztendlich miteinander verschmelzen. Dieser Vorgang wird auch als Koaleszenz bezeichnet. Für diese Art von Problemen existiert keine analytische Lösung, weshalb hier nur das quantitative Verhalten bewertet werden kann.

Da es sich um ein instationäres Problem handelt bei dem sich die geometrische Position des Phaseninterfaces stetig ändert, eignet es sich zur Veranschaulichung der parallelen, adaptiven Gitterstruktur. Der Prototyp von VIRTUALFLUIDS berücksichtigte zum Zeitpunkt der Simulation adaptiv, parallel ausschließlich zwei verschiedene Gitterlevel. In diesem Beispiel hatte das umgebende Fluid eine Dichte von 3,0, die zweite Phase die Dichte 2,0, die Viskosität beider Phasen betrug 1/10 und die Oberflächenspannung wurde zu 1/100 gesetzt.

Das Startgitter hatte insgesamt $8 \cdot 10^6$ Berechnungsknoten, wobei jeder Block 9^3 Knoten enthielt. Die Anpassung des Gitters erfolgte in jedem zwanzigsten groben Zeitschritt mit den in Abs. 17.7.5 beschriebenen Gitterbesucherklassen. Die Verfeinerungs- und Vergröberungsparamter wurden entsprechend definiert. Der kritische Bereich der Knotenmatrix betrug z. B. drei Knoten zum Blockrand während der innere Überprüfungsbereich bis hin zu fünf Knoten Entfernung zum Rand Anwendung fand. Der von der Gitteradaptvität bei einer Berechnung mit 35 Rechendiensten beanspruchte Anteil an der Gesamtrechenzeit in Höhe von ungefähr sieben Prozent kann im Vergleich zum benötigten Mehraufwand einer uniformen Berechnung des gleichen Problems vernachlässigt werden.



(a) u_{x_2} ***(b)*** u_{x_3}

Abbildung 17.38: *Isolinien der Geschwindigkeitskomponenten (Schnitt)*

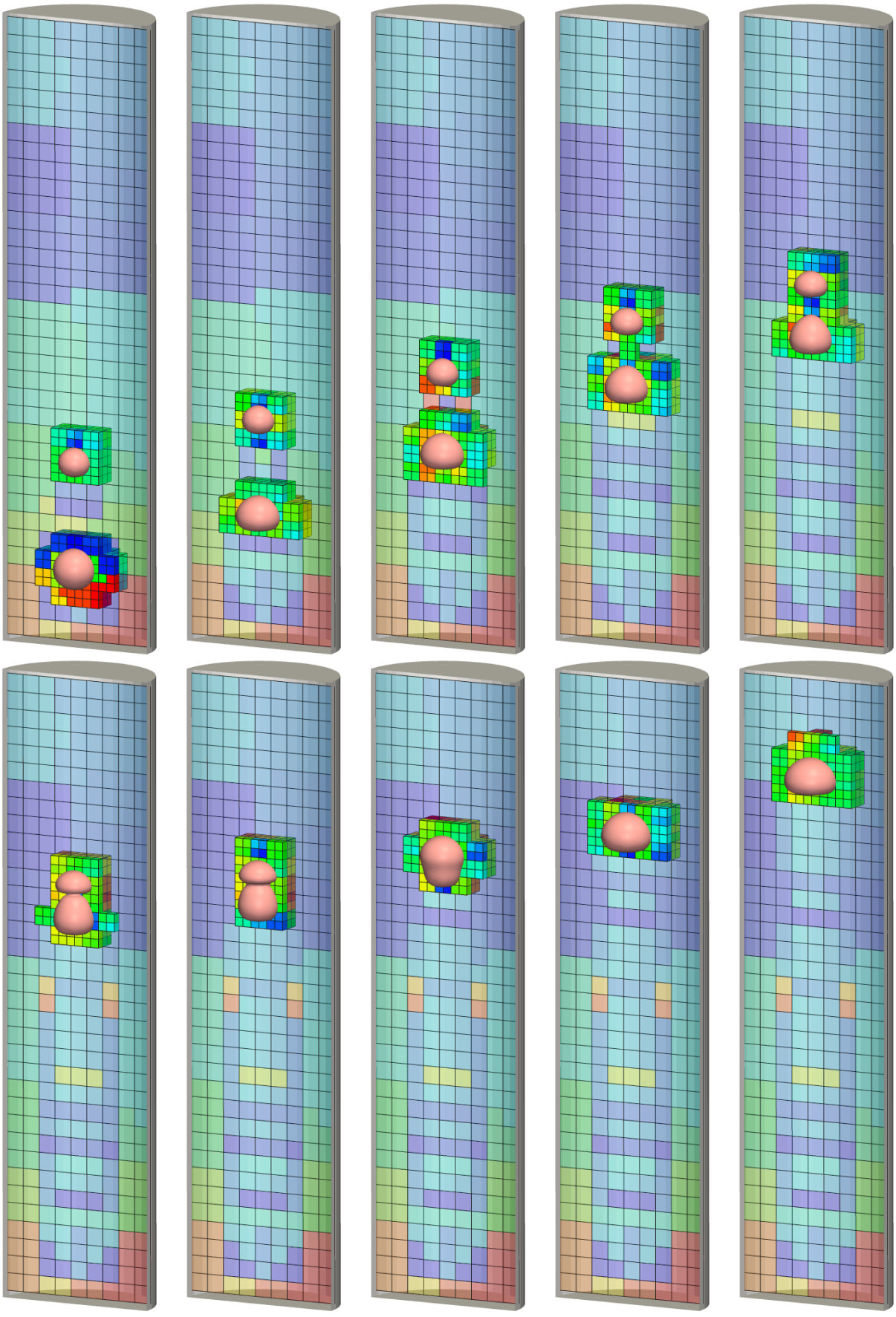


Abbildung 17.39: *Zwei hintereinander aufsteigende Blasen auf adaptivem Gitter (farbig: Gebietszerlegung)*

Die Simulationsergebnisse entsprechen denen einer seriellen Berechnung und dem erwarteten physikalischen Verhalten. In Abb. 17.38 sind exemplarisch die Isolinien für verschiedene Geschwindigkeitskomponenten der aufsteigenden Blasen über verschiedene Gitterlevel hinweg dargestellt. Die Bildsequenz in Abb. 17.39 zeigt die Isokonturen der beiden Blasen, die jeweils veränderte Gitterstruktur und die Prozessverteilung zu verschiedenen Zeitpunkten der Simulation. Aus Übersichtsgründen ist jeweils nur die hintere Hälfte des zur x_3-Achse achsensymmetrischen Gitters dargestellt. Hierbei lässt sich gut erkennen, dass das Gitter hauptsächlich in Strömungsrichtung verfeinert wird, orthogonal dazu werden weniger Blöcke vorgehalten und der Nachlauf der Blasen wird frühzeitig vergröbert. Dies ist ein eindeutiger Vorteil gegenüber dem Knotencode, bei dem die Umsetzung einer solchen Abschätzung aufgrund der geometrischen Komplexität wesentlich aufwändiger ist. Wie bereits in Abs. 17.7.5 erwähnt, erfolgte die Prozessverteilung der neu entstehenden Blöcke durch den TopologyService mittels eines heuristischen Ansatzes. Der resultierende Interprozesskommunikationsaufwand ist somit noch nicht optimal, da benachbarte Blöcke oftmals verschiedenen Prozessen zugewiesen wurden. Der Arbeitsaufwand der einzelnen Prozesse hingegen war gleichwertig.

17.10.4 Schallabsorption durch poröse Medien

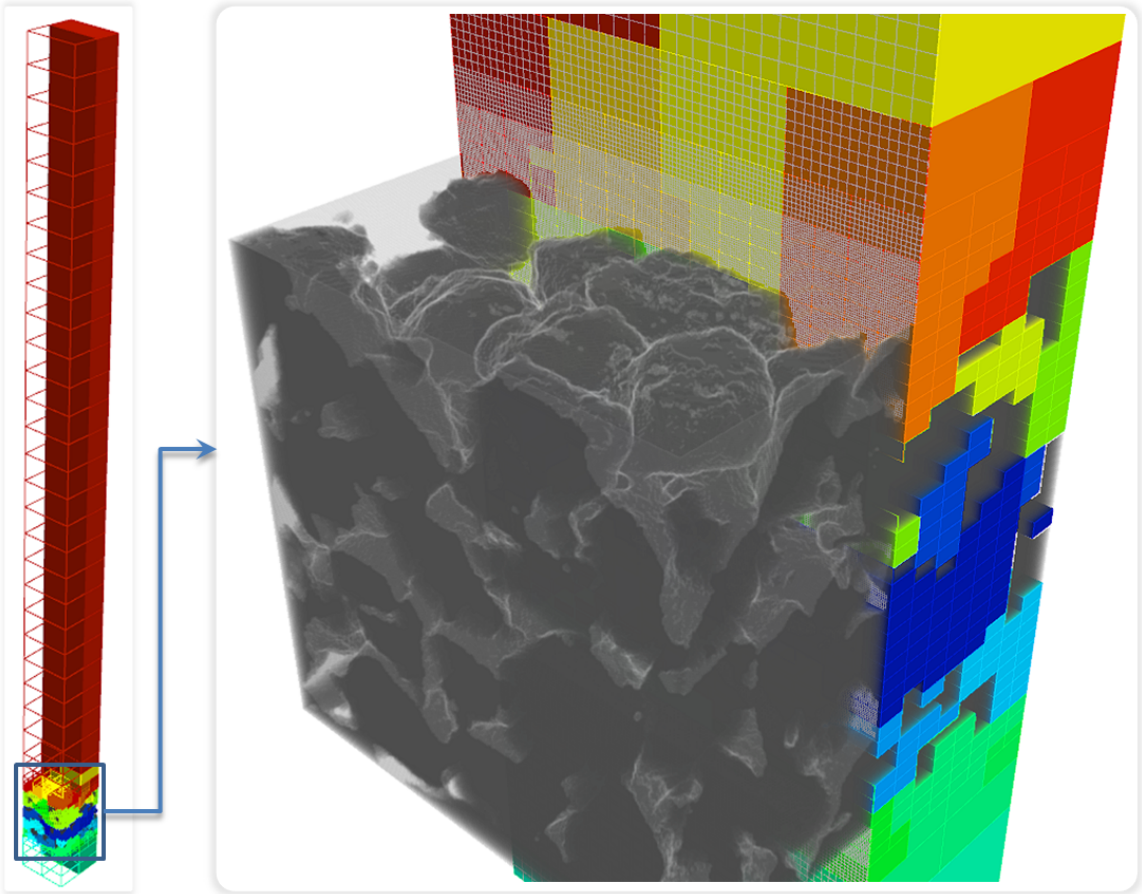
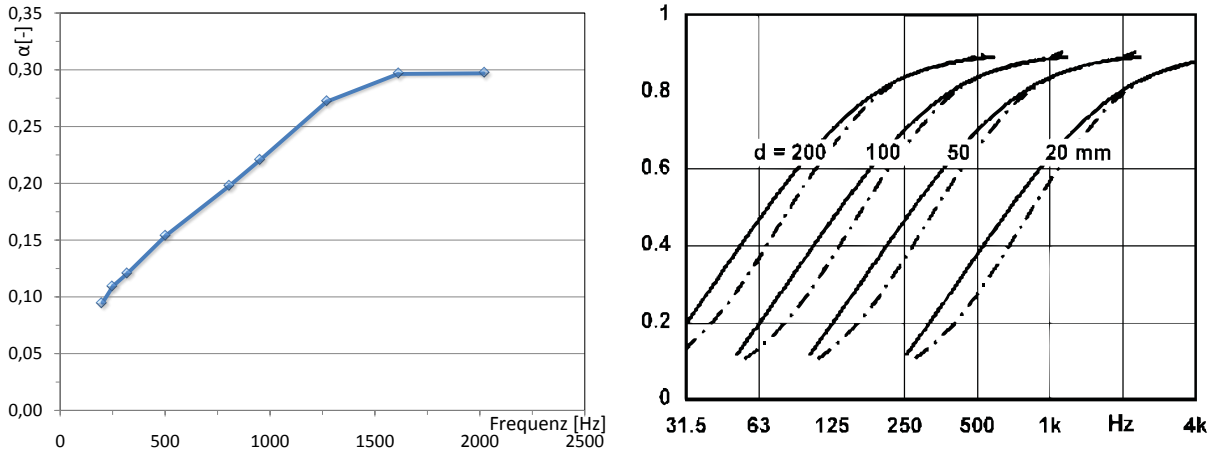


Abbildung 17.40: *Gitterkonfiguration mit METIS-Zerlegung*

In diesem Beispiel wird mit VIRTUALFLUIDS anhand der Bestimmung der Schalldämpfung eines porösen Straßenbelags die Akustik in porösen Medien simuliert. Auf die physikalischen Herausforde-

rungen dieses Testfalls und der Anwendbarkeit des Lattice-Boltzmann-Verfahrens für Akustik wird ausführlich in der Dissertation von Benjamin Ahrenholz [3] eingegangen, der die hier vorgestellten Berechnungen unter anderem mit VIRTUALFLUIDS durchführte.

Die als Topographiescan zur Verfügung stehende Bodenprobe einer grobporigen Asphaltdecke umfasste in jede Raumrichtung mindestens fünf Poren sowie fünf Körner und wurde zur Auflösung der akustischen Grenzschicht mit 250^3 Knoten diskretisiert. Die für die Berechnung benötigte Luftsäule oberhalb des porösen Mediums war abhängig von der Wellenlänge der jeweiligen zu untersuchenden Schallfrequenz. Während die erforderliche Höhe bei 1.000 Hz ≈ 3.500 Knoten mit dem Diskretisierungsabstand der Bodenprobe betrug, wurden bei 100 Hz bereits ≈ 35.000 Knoten benötigt, was bei einer uniformen Auflösung $2,2 \cdot 10^9$ Gitterknoten erfordert hätte. Durch die Verwendung der nichtuniformen Blockstruktur konnte die Anzahl an Freiheitsgraden auf ein Minimum reduziert werden (Abb. 17.40). Für den Testfall mit 100 Hz war die tatsächlich verwendete Knotenanzahl drei Größenordnungen kleiner.



(a) *Berechnete Absorptionskoeffizienten*

(b) *Typische Absorptionskoeffizienten für poröse Absorber [41]*

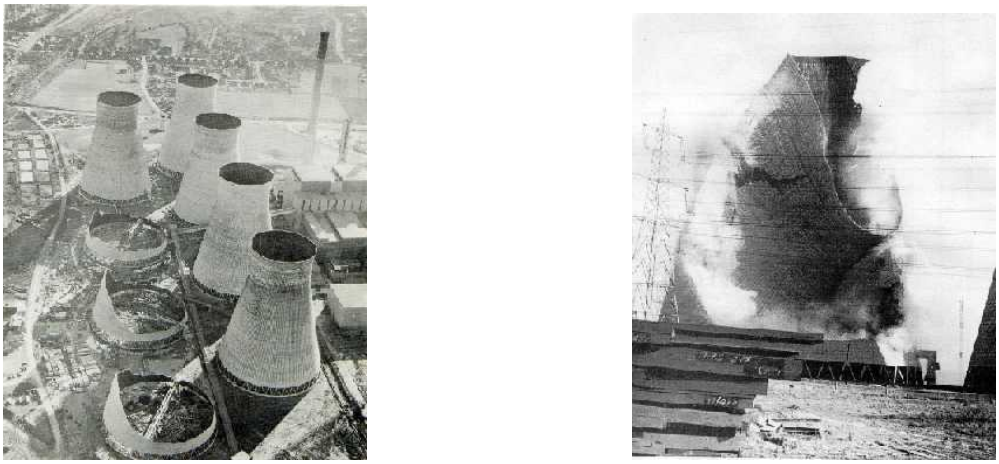
Abbildung 17.41: *Absorptionskoeffizienten*

In der Simulationen wurde die Generierung der Schallwellen mit Hilfe einer pulsierenden Einflussdruckrandbedingung realisiert. Für die Ermittlung der Dämpfung wurde eine zeitabhängige Besucherklasse verwendet, die die Druckentwicklung für verschiedene Messpunkte aufzeichnete. Die Absorptionsergebnisse sind in Abb. 17.41a dargestellt und sind vom Verlauf her ähnlich zu den Referenzkurven inAbb. 17.41b. Allerdings bestehen hohe Unterschiede in der Amplitude. Die Ursache hierfür ist laut [3] unter anderem der fehlende Resonanzeffekt innerhalb der Bodenprobe. Eine ausführliche Diskussion diesbezüglich ist in [3] zu finden. Für zufriedenstellende Ergebnisse besteht weiterer Forschungsbedarf.

17.10.5 Kühlturmumströmung

Strömungssimulationen gewinnen im Bauwesen immer mehr an Bedeutung. Bei hohen Windlasten können z. B. Resonanzeffekte durch sich überlagernde Luftwirbel auftreten, die ernsthafte Auswirkungen auf die Standfestigkeit von Gebäuden haben können. Eines der bekanntesten Beispiele aus

dem Bauingenieurbereich sind die im Jahr 1965 durch Windkraft zerstörten Kühltürme bei Ferrybridge in England (Abb. 17.42).



(a) *Luftansicht* ***(b)*** *Einstürzender Kühlturm*

Abbildung 17.42: *Ferrybridge Kühltürme (1965) [112]*

Für realistische Simulationen von derartigen Strömungen sind Reynoldszahlen in der Größenordnung von $Re = \mathcal{O}(10^8)$ erforderlich und liegen außerhalb des Limits für Standard-LES Simulationen. Die Verwendung von dynamischen LES-Modellen oder RANS-Turbulenzmodellen könnte hier Abhilfe schaffen. Um bei Kühltürmen das prinzipielle Strömungsverhalten bei hohen Windgeschwindigkeiten zu veranschaulichen, wurden zwei benachbarte Kühltürme unter Verwendung des in Abs. 17.10.1 beschriebenen LES-Turbulenzmodells für $Re = 100.000$ berechnet. Hierfür wurde das Blockgitter mit $\approx 5 \cdot 10^8$ Freiheitsgraden auf 20 Rechendienste verteilt und $3 \cdot 10^5$ Mal iteriert. Für diese 20 Minuten Echtzeit benötigte die Simulation auf dem iRMB-PC-Cluster $\approx$ 120 Stunden.

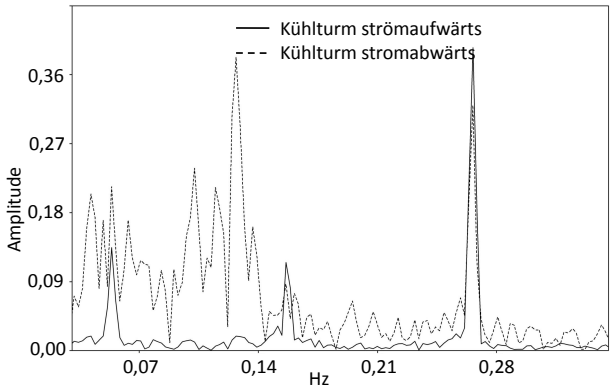


Abbildung 17.43: *Kraftspektrum der Druckkraft für beide Türme*

Während des letzten Viertels der Simulation wurde die Druckkraft auf die beiden Kühltürme kontinuierlich gemessen. Wie in Abb. 17.43 zu sehen, weisen die Kraftspektren der beiden Türme deutliche Unterschiede auf, was bei der Planung derartiger Gebäude eine wesentliche Rolle spielt. Für konkretere Aussagen bedarf es weiterer, eingehender Simulationen mit verbesserter Turbulenzmodellierung sowie umfassender Vergleiche mit experimentellen Daten

An dem Beispiel soll verdeutlicht werden, dass heutzutage unter Verwendung von computergestützter Modellierung wichtige Aussagen über Einfluss von Strömungen selbst auf große Strukturen getroffen werden können. Eine solche Simulation kann zukünftig auch von kleinen Ingenieurbüros vergleichsweise kostengünstig gegenüber Windkanalversuchen unter Verwendung von GPUs [155] durchgeführt werden. Abb. 17.44 zeigt eine Momentaufnahme des Geschwindigkeitsfeldes zu einem fortgeschrittenen Zeitpunkt der Simulation.

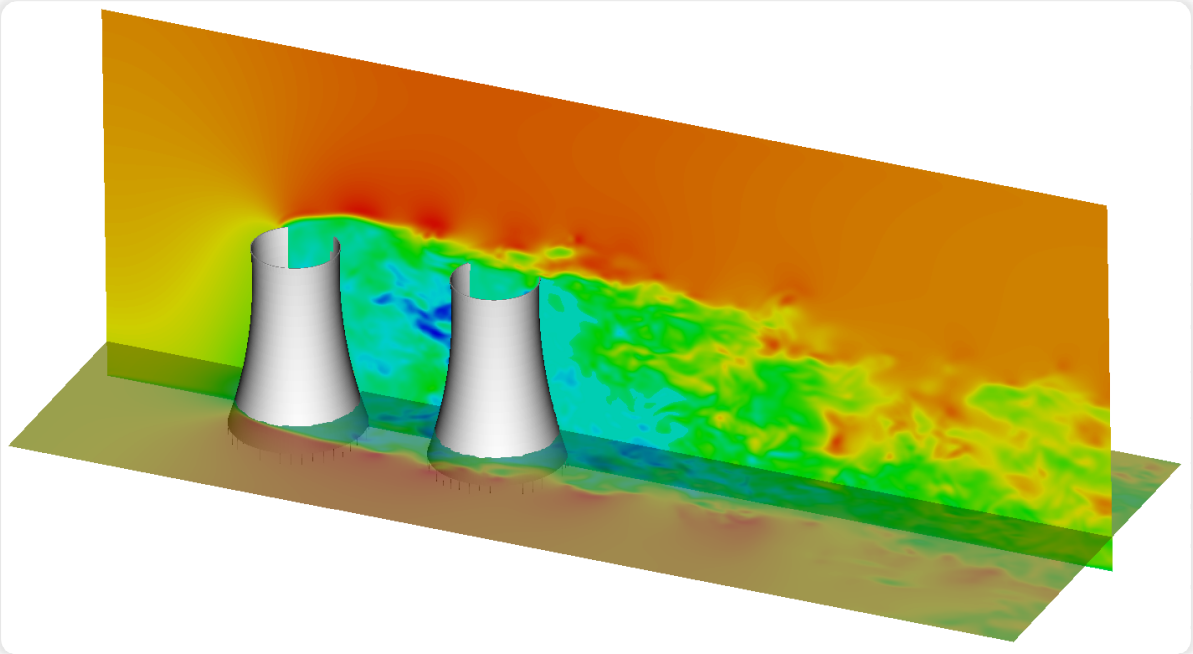


Abbildung 17.44: *Geschwindigkeitsfeld einer turbulenten Kühltrumumströmung bei* $Re = 100.000$

18 Zusammenfassung und Ausblick

18.1 Zusammenfassung

Ziel der Arbeit war die Entwicklung eines Konzeptes, mit dem unterschiedliche Aspekte, die bei der Berechnung von CFD-Problemen auftreten, in einem Code effizient und flexibel untergebracht werden können. Die numerische Basis für die entwickelten Softwarekonzepte und deren Umsetzung stellte das Lattice-Boltzmann-Verfahren dar, dass in der beschriebenen Finite-Differenzen-Diskretisierung auf nicht-uniformen, kartesischen Baumgittern Anwendung findet. Für die in diesem Zusammenhang präsentierte Erweiterung des Rothmann-Keller-Modells zur Simulation von Mehrphasenströmungen erwies sich die Verwendung einer dynamisch, adaptiven Gitterstruktur aufgrund numerischer Störungen bei Überschneidung des physikalischen und des numerischen Interfaces als vorteilhaft.

Das entwickelte baumbasierte Knotengitterkonzept hat zum Vorteil, dass durch die strikte Trennung von Topologie und Physik und durch die Verwendung objektorientierter Programmiertechniken der Großteil der Programmfunktionalität generalisiert zur Verfügung gestellt werden kann. Somit können dem Löser mit verhältnismäßig geringem Aufwand weitere numerische Kerne hinzugefügt werden. Verschiedene, im Detail erläuterte Erweiterungen der Basisstruktur ermöglichen sowohl adaptive als auch verteilte Simulationen. Letztere sind vor allem in Hinblick auf die Verwendbarkeit des Softwarepaketes auf PC-Clustern mit verteiltem Speicher aber auch modernen Mehrkern-CPUs von großer Bedeutung. In diesem Zusammenhang wurde ein generalisiertes Regelframework erstellt, mit dessen Hilfe das der Simulation zugrunde liegende Berechnungsgitter für unterschiedliche, benutzerdefinierte Anforderungen auf verschiedene Prozesse verteilt und berechnet werden kann. Bei der Implementierung wurde darauf geachtet, dass diese Erweiterung mit minimalem Aufwand für bereits bestehende, serielle Löser des Knotencodes verwendet werden kann.

Anhand verschiedener Validierungsrechnungen wurde die Eignung des LB-Verfahrens für die Simulation von Ein- und Mehrphasenproblemen sowie die Notwendigkeit nicht-uniformer Berechnungsgitter aufgezeigt. Bei der Evaluierung des vorgestellten Mehrphasenmodells anhand von Standardtestfällen, wie Zweiphasenspaltströmungen oder aufsteigenden Blasen, kamen u. a. dynamisch adaptive Gitter zum Einsatz. Eine Konvergenzstudie bestätigte dem Modell eine räumliche Genauigkeit zweiter Ordnung. Die Messung verschiedener, für verteilte Löser relevanter Kenngrößen, wie die Skalierung und die parallele Effizienz, sowie die Simulation einer Tandemzylinderumströmungen bestätigten das Potential des parallelen Lösers.

Die hohe Flexibilität der Datenstruktur implizierte einige Nachteile, wie einen verhältnismäßig hohen Speicherbedarf sowie eine erhöhte Komplexität bezüglich der Umsetzung dynamisch adaptiver Berechnungsgitter. Da für letzteres die mangelnde Unterstützung einer dynamischen Kommunikation seitens MPI eines der Hauptprobleme darstellte, wurden für das zweite Softwarekonzept verschiedene Interprozessbibliotheken, wie CORBA und Ice, auf deren Verwendbarkeit in VIRTUALFLUIDS untersucht. Aufgrund der verhältnismäßig einfachen Anwendbarkeit fiel die Entscheidung zugunsten des Remote Call Frameworks (RCF) von Jarl Lindrud, das durch die Unterstützung des Client/Server-

Musters die geforderte dynamische Kommunikation und durch das zur Verfügung gestellte Serialisierungsframework ein einfaches Versenden komplexer Objekte ermöglicht.

Der neu konzipierte, hybride Blockgitteransatz, bei dem die Trennung von Topologie und Physik beibehalten wurde, ist eine konsequente Weiterentwicklung des Knotengitteransatzes. Hierbei wird die Baumstruktur für ein Blockgitter erstellt, für das im Wesentlichen die gleichen Anforderungen gelten wie bereits für das Knotengitter. Anstelle der Repräsentation der Knoten durch eigenständige Objekte, werden deren Informationen in gleichförmigen Matrizen innerhalb der einzelnen Blöcke vorgehalten. Dies ermöglicht eine effiziente Berechnung bei zugleich optimalem Speicherbedarf. Die Kommunikation zwischen den Blöcken erfolgt mit dem eingeführten Connector-Transmitter-Konzept, bei dem die Kommunikation autark mit verschiedenen Modulen lokal oder verteilt erfolgen kann. Verteilte Berechnungen werden beim Blockgitter mit Hilfe servicebasierter, nach Aufgabengebieten getrennter Komponenten durchgeführt, die auf Basis des RCF umgesetzt wurden. In diesem Zusammenhang wurden verschiedene Funktionalitäten für verteilte Anwendungen, die für einen Multiphysik-Löser von Bedeutung sind, detailliert beschrieben und umgesetzt. Zu diesen gehören unter anderem die Unterstützung von komplexen Geometrieobjekten, die Möglichkeit einer Fluid-Struktur-Interaktion sowie eine dynamisch adaptive, verteilte Gittersteuerung. Die numerische Kommunikation zwischen den Rechendiensten kann dabei mit Hilfe des Connector-Transmitter-Konzeptes entkoppelt von den Services über eine beliebige Interprozesskomunikationsbibliothek erfolgen.

Der Ansatz wurde anhand diverser Simulationen eingehend hinsichtlich seiner parallelen Effizienz und Skalierbarkeit untersucht und wies insgesamt eine gute Leistung auf. In diesem Zusammenhang wurden auch verschiedene Interprozessprotokolle sowie Datenaustauschmodule in Bezug auf ihre Leistungsfähigkeit untersucht. Dabei erwies sich der entwickelte Pooltransmitter, der die gesammelten numerischen Informationen mehrerer Blöcke in einem Schritt mit dem Nachbarprozess austauscht, als sehr effizientes Übertragungsmodul. Abgerundet wurde dieser Teil durch die Präsentation verschiedener Anwendungsbeispiele, bei denen der hybride, verteilt arbeitende Blockcode zum Einsatz kam. Diese verdeutlichten erneut, dass ein parallel arbeitender, nicht-uniformer Strömungslöser unabdingbar für effiziente Simulationen realer Problemstellungen ist.

18.2 Ausblick

Die in dieser Arbeit vorgestellten verteilten Ansätze wurden primär für PC-Cluster und Grid-Computing mit wenigen hundert Rechenkernen entwickelt. Für massiv-parallele Systeme mit mehreren tausend Rechenkernen ist der beschriebene Master-Slave-Ansatz in der jetzigen Form nur bedingt einsetzbar, da es bei einer sehr hohen Anzahl zu verwaltender Rechendienste zu Speicherengpässen und Performanceproblemen seitens des jeweiligen topologiehaltenden Prozesses kommen kann. Denkbare Lösungen für dieses Problem sind entweder dessen Parallelisierung oder die Übertragung seiner Aufgaben auf die einzelnen Rechendienste.

Ein weiteres nicht vollständig gelöstes Problem ist die Optimierung der Gebietszerlegung für nicht-uniforme und vor allem dynamisch adaptive Gitter. Die Verwendung der graphenbasierten METIS-Bibliothek zur Gebietszerlegung erzielte bei uniformen Gittern und entsprechend gewählten Wichtungen eine sehr ausgewogene Lastverteilung. Bei nicht-uniformen Gittern ist das Resultat oft unzureichend, da eine Einflussnahme durch den Benutzer nur sehr eingeschränkt möglich ist. So kann beispielsweise die Trennung bestimmter im Graphen enthaltener Kanten nicht untersagt werden. Aus diesem Grund ist die Untersuchung oder Entwicklung alternativer Zerlegungsalgorithmen notwen-

dig, die im optimalen Fall ähnlich zum Regelframework bei verteilten Knotengittern benutzerdefinierbare Konditionen berücksichtigt.

Gegenwärtig wird VIRTUALFLUIDS um einen Berechnungskern zur Simulation von freien Oberflächen erweitert, die durch einen VOF-Ansatz beschrieben werden. Ähnlich zu Mehrphasenströmungen ist dabei eine adaptive Gitterstruktur von Vorteil. In diesem Zusammenhang ist das Erzeugen neuer Blöcke zur Laufzeit eine Herausforderung für die verteilte servicebasierte Softwarestruktur. Erste Ergebnisse sind sehr vielversprechend. Ebenfalls Gegenstand aktueller Forschungen ist die Entwicklung und Implementierung von Wandmodellen sowie nicht reflektierender Randbedingungen für weiterführende realitätsgetreue Turbulenzsimulationen mit hohen Reynoldszahlen sowie die Umsetzung einer effizienten Fluid-Struktur-Interaktion für eine große Anzahl geometrischer Körper.

Anlagen

Anhang

A1 Orthogonale Basisvektoren

Orthogonale Basisvektoren $\mathbf{\Phi_i}$ für das $d3q19$-Modell:

$$
\begin{aligned}
\Phi_{0,\alpha} &= 1 \\
\Phi_{1,\alpha} &= \mathbf{e}_\alpha^2 - c^2 \\
\Phi_{2,\alpha} &= 3\left(\mathbf{e}_\alpha^2\right)^2 - 6\,\mathbf{e}_\alpha^2\,c^2 + c^4 \\
\Phi_{3,\alpha} &= e_{\alpha x_1} \\
\Phi_{5,\alpha} &= e_{\alpha x_2} \\
\Phi_{7,\alpha} &= e_{\alpha x_3} \\
\Phi_{4,\alpha} &= \left(3\,\mathbf{e}_\alpha^2 - 5\,c^2\right)\, e_{\alpha x_1} \\
\Phi_{6,\alpha} &= \left(3\,\mathbf{e}_\alpha^2 - 5\,c^2\right)\, e_{\alpha x_2} \\
\Phi_{8,\alpha} &= \left(3\,\mathbf{e}_\alpha^2 - 5\,c^2\right)\, e_{\alpha x_3} \\
\Phi_{9,\alpha} &= 3\,e_{\alpha x_1}^2 - \mathbf{e}_\alpha^2 \\
\Phi_{10,\alpha} &= \left(2\,\mathbf{e}_\alpha^2 - 3\,c^2\right)\left(3e_{\alpha x_1}^2 - \mathbf{e}_\alpha^2\right) \\
\Phi_{11,\alpha} &= e_{\alpha x_2}^2 - e_{\alpha x_3}^2 \\
\Phi_{12,\alpha} &= \left(2\,\mathbf{e}_\alpha^2 - 3\,c^2\right)\left(e_{\alpha x_2}^2 - e_{\alpha x_3}^2\right) \\
\Phi_{13,\alpha} &= e_{\alpha x_1}\, e_{\alpha x_2} \\
\Phi_{14,\alpha} &= e_{\alpha x_2}\, e_{\alpha x_3} \\
\Phi_{15,\alpha} &= e_{\alpha x_1}\, e_{\alpha x_3} \\
\Phi_{16,\alpha} &= \left(e_{\alpha x_2}^2 - e_{\alpha x_3}^2\right) e_{\alpha x_1} \\
\Phi_{17,\alpha} &= \left(e_{\alpha x_3}^2 - e_{\alpha x_1}^2\right) e_{\alpha x_2} \\
\Phi_{18,\alpha} &= \left(e_{\alpha x_1}^2 - e_{\alpha x_2}^2\right) e_{\alpha x_3}
\end{aligned}
$$

A2 Transformationsmatrix

Die verwendete Transformationsmatrix M für das $d3q19$-Modell lautet:

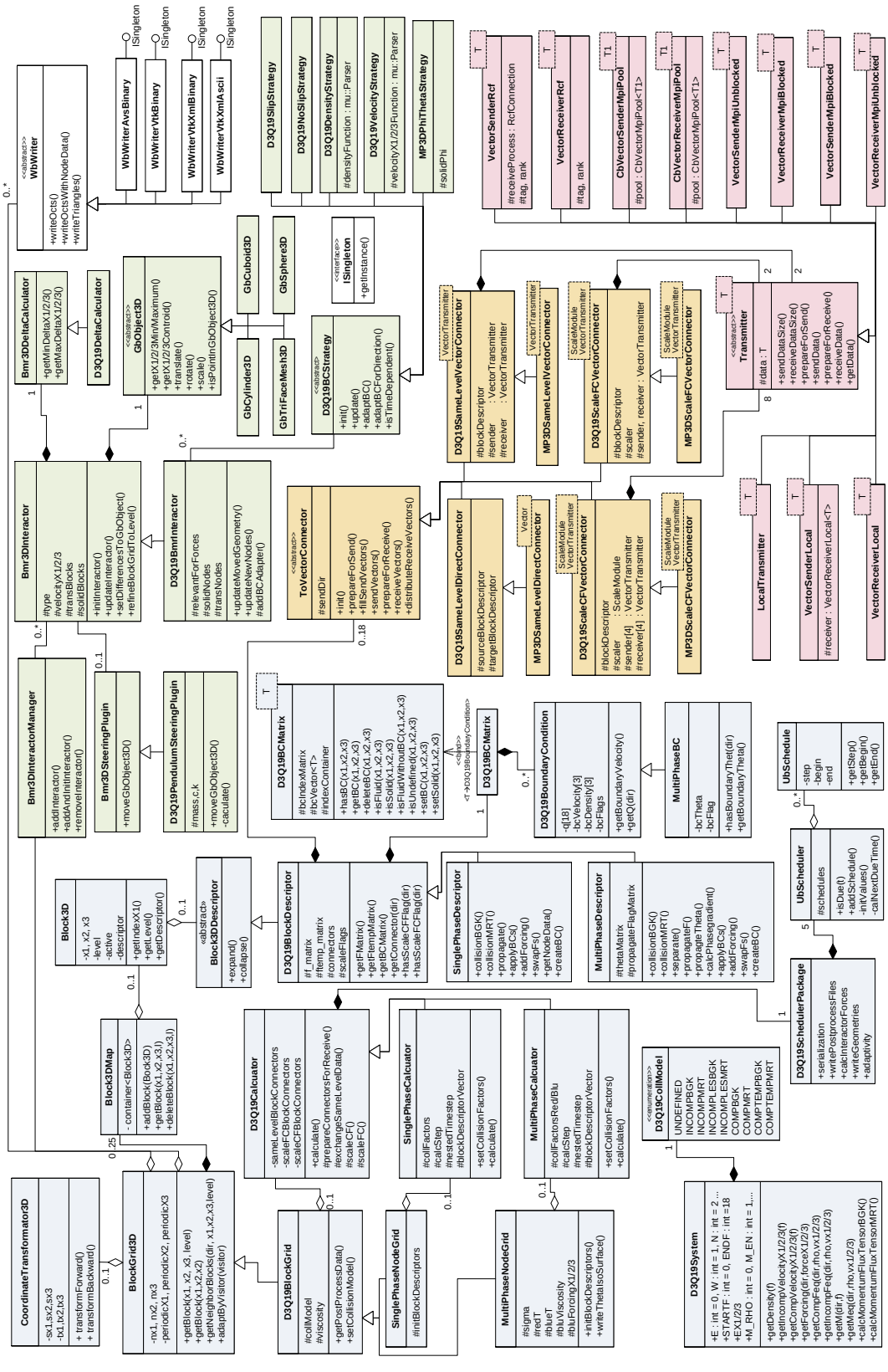
$$\begin{bmatrix}
1\cdot & (1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1) \\
c^2\cdot & (-1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1) \\
c^4\cdot & (1 & -2 & -2 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1) \\
c\cdot & (0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0) \\
c^3\cdot & (0 & -2 & 2 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0) \\
c\cdot & (0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1) \\
c^3\cdot & (0 & 0 & 0 & -2 & 2 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1) \\
c\cdot & (0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1) \\
c^3\cdot & (0 & 0 & 0 & 0 & 0 & -2 & 2 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1) \\
c^2\cdot & (0 & 2 & 2 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -2 & -2 & -2 & -2) \\
c^4\cdot & (0 & -2 & -2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -2 & -2 & -2 & -2) \\
c^2\cdot & (0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0) \\
c^4\cdot & (0 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0) \\
c^2\cdot & (0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) \\
c^2\cdot & (0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1) \\
c^2\cdot & (0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0) \\
c^3\cdot & (0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0) \\
c^3\cdot & (0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1) \\
c^3\cdot & (0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1)
\end{bmatrix}$$

A3 Geometrische Richtungen

Index	0	1	2	3	4	5	6	7		
r	E	W	N	S	NE	SW	SE	NW		
Index	8	9	10	11	12	13	14	15	16	17
r	T	B	TE	BW	BE	TW	TN	BS	BN	TS
Index	18	19	20	21	22	23	24	25		
r	TNE	TNW	TSE	TSW	BNE	BNW	BSE	BSW		

Tabelle A.1: *Deklaration der Raumrichtungen r_i (E=East, W=West, N=North, S=South, T=Top, B=Bottom)*

A4 Blockgitter Softwarestruktur



A5 Implementierung

In diesem Abschnitt wird die Umsetzung einiger Klassen von VIRTUALFLUIDS in C++ sowie die Realisierung einer RCF-Server-Client-Anwendung genauer beschrieben.

A5.1 Transmitter

Ein Transmittermodul (vgl. Abschn. 15.1) hält das zu transferierende Datenelement data direkt in der Basisklasse (Quellcode A.1). Dies hat zum Vorteil, dass der Compiler Datenzugriffe durch statische Bindung optimieren kann. Der Einsatz von Polymorphie hätte zur Folge, dass bei jedem Datenzugriff mit Hilfe der *Tabelle virtueller Methoden* (*engl.:* vtable) zunächst eine zeitintensive Ermittlung der konkreten Methode erforderlich wäre (dynamische Bindung).

Quellcode A.1: *Klasse Transmitter*

```
1  template<typename value_type>
2  class Transmitter
3  {
4  public:
5    Transmitter() { }
6    virtual ~Transmitter() { }
7
8    //Zugriff auf die Daten
9    inline       value_type&  getData()       { return this->data; }
10   inline const value_type&  getData() const { return this->data; }
11
12   //Information "local"/"remote"
13   virtual bool isLocalTransmitter()  const = 0;
14   virtual bool isRemoteTransmitter() const { return !this->isLocalTransmitter(); }
15
16   //Präprozess (z.B. zum Abgleich der Sende- und Empfangspuffer)
17   virtual void sendDataSize()      { /* empty */ }
18   virtual void receiveDataSize()   { /* empty */ }
19
20   //Methoden für Datentransfer
21   virtual void prepareForSend()    { /* empty */ }
22   virtual void prepareForReceive() { /* empty */ }
23   virtual void        sendData()    = 0;
24   virtual value_type& receiveData() = 0;
25
26 protected:
27   value_type data;
28 };
```

Derzeit verwendet VIRTUALFLUIDS ausschließlich Transmitterobjekte mit Vektoren als Datenelement. Prinzipiell kann jede Vektorklasse verwendet werden, die über bestimmte Basisfunktionalitäten eines Vektors der *Standard Template Library (STL,* [164]*)* verfügt. VIRTUALFLUIDS stellt hierfür zusätzlich die CbVector-Klasse zur Verfügung, die u. a. bei dem von den Pooltransmittern (Abs. 17.8.3) verwendeten Vektorpool von Bedeutung ist.

Die wesentlichen Methoden eines Transmitters sind:

- sendDataSize/receiveDataSize
  Diese werden vorwiegend im Präprozess von verteilten Berechnungen für den Längenabgleich der später auszutauschenden Datenvektoren verwendet
- prepareForSend/prepareForReceive
  Für Aufgaben, die jeweils vor dem Senden bzw. Empfangen durchgeführt werden müssen,

werden diese Methoden angewendet. Bei einer asynchronen Kommunikation kommen hier Schutzmechanismen für die Datenpuffer, bei MPI z. B. in Form von request-Objekten, zum Einsatz, die verhindern, dass noch benötigte Puffer vorzeitig überschrieben werden.

- sendData und receiveData
  Hier erfolgt der eigentliche Datenaustausch zwischen den Transmittermodulen.

Für den lokalen Datenaustausch stehen dem Anwender zwei Transmittertypen zur Verfügung. Der einfachere und sehr effiziente ist der LocalTransmitter (Quellcode A.2). Zwei benachbarte Connectoren bekommen dieselbe Instanz dieser Klasse zugewiesen und arbeiten anschließend auf dem dort enthaltenen Datenelement (z. B. value_type=std::vector<double>). Da beide Seiten mit derselben Komponente arbeiten, wird keine Übertragungslogik benötigt.

Quellcode A.2: *Klasse LocalTransmitter*

```
1  template<typename value_type>
2  class LocalTransmitter : public Transmitter< value_type >
3  {
4  public:
5    LocalTransmitter() : Transmitter< value_type >()  { }
6
7    bool isLocalTransmitter()   { return true;       }
8    void sendDataSize()         { /*do nothing*/     }
9    void receiveDataSize()      { /*do nothing*/     }
10   void sendData()             { /*do nothing*/     }
11   value_type&  receiveData()  { return this->data; }
12 };
```

Das zweite, vorwiegend für Diagnosezwecke genutzte Transmittertupel besteht aus dem VectorSenderLocal (Quellcode A.4) und dem VectorReceiverLocal (Quellcode A.3). Beim Datentransfer werden die Informationen zwischen dem Senderobjekt des übermittelnden Connectors und dem passenden Empfangsmodul des entgegennehmenden Connectors ausgetauscht. Entweder kopiert der Sender dafür beim Senden die Daten in das Datenelement des entsprechenden Empfängers, oder der Empfänger holt sich dieses beim Empfangen vom zugehörigen Sender. Die hier abgebildete Version nutzt erstere Variante.

Quellcode A.3: *Klasse VectorReceiverLocal*

```
1  template<typename value_type>
2  class VectorReceiverLocal : public Transmitter< value_type >
3  {
4  public:
5    VectorReceiverLocal() : Transmitter< value_type >() { }
6
7    bool isLocalTransmitter()  const { return true; }
8
9    void sendDataSize()  { throw UbException("receives only"); }
10   void sendData()      { throw UbException("receives only"); }
11
12   void         receiveDataSize() { /*do nothing*/     }
13   value_type&  receiveData()     { return this->data; }
14 };
```

Diese Vorgehensweise entspricht der von RemoteTransmittern. Zum Senden und Empfangen verwenden diese die entsprechende Methoden der jeweiligen Interprozessbibliothek. Die hierfür benötigten Informationen werden diesen während der Initialisierung durch das SetConnectorVisitor-Modul zugewiesen.

Quellcode A.4: *Klasse VectorSenderLocal*

```
1  template<typename value_type>
2  class VectorSenderLocal : public Transmitter< value_type >
3  {
4  public:
5    VectorSenderLocal(SmartPtr<VectorReceiverLocal<value_type> > receiver)
6       : Transmitter< value_type >(), receiver(receiver)
7    {
8    }
9
10   bool isLocalTransmitter()  const { return true; }
11
12   void sendDataSize()  { receiver->getData().resize( this->data.size() ); }
13   void sendData()      { receiver->getData() = this->data;                 }
14
15   void         receiveDataSize() { throw UbException("sends only"); }
16   value_type&  receiveData()     { throw UbException("sends only"); }
17
18 protected:
19   SmartPtr< VectorReceiverLocal< value_type > > receiver;
20 };
```

A5.2 Generische Programmierung (Template, Traits)

Im Folgenden wird beispielhaft anhand der Klasse D3Q19AssignLocalConnectorsVisitor gezeigt, wie durch Verwendung generischer Programmierung mit Templates und Traits größtmögliche Flexibilität erreicht werden kann. Diese Klasse wird für die Zuweisung der lokalen Blockconnectoren und Transmitter verwendet. Bei den Verbindungsmodulen kommen, wie in Abschn. 15.2 beschrieben, in Abhängigkeit der Gitterlevel zweier benachbarter Blöcke SameLevelConnectoren, ScaleCFConnectoren oder ScaleFCConnectoren zum Einsatz. Die Ermittlung des zu verwendenden Typs ist dabei verhältnismäßig aufwändig, jedoch unabhängig vom zu lösenden Strömungsproblem (Einphase, Mehrphase, etc.), Kollisionsmodell (BGK, MRT) und den zu verwendenden Transmittern (LocalTransmitter, VectorSender/ReceiverLocal, etc).

Ein klassischer Lösungsansatz würde in solchen Fällen eine Implementierung für eine Vielzahl von Modulkombinationen zur Verfügung stellen. Eine solche Vorgehensweise resultiert allerdings in redundantem und schwer zu wartendem Quellcode. Zudem ist eine separate Implementierungen aller möglichen Kombinationen nahezu unmöglich.

Die optimale Lösung für derartige Probleme stellt die Verwendung von Templates [161] dar, mit denen vom Datentyp unabhängige Programmgerüste zur Verfügung gestellt werden können. Da beim D3Q19AssignLocalConnectorsVisitor eine Vielzahl an Typen verwendet wird, bietet sich zudem die Verwendung sogenannter Traits [161] an, die eine weitere wichtige Technik bei der generischen C++ Programmierung darstellen. Eine Traitklasse stellt dabei die Charakteristiken eines Typs zur Verfügung, die ein generischer Algorithmus (z. B. Klassentemplate) für die korrekte Behandlung des Typs benötigt, aus dem Typ selbst aber nicht ermittelt werden können. Mit Charakteristiken sind dabei sowohl Daten (z. B. Konstanten und Typen) als auch Verhaltensweisen (Funktionen) gemeint.

Das D3Q19AssignLocalConnectorsVisitor-Klassentemplate benötigt beispielsweise ein ModelTrait, in dem u. a. der Blockgittertyp sowie die zu verwendenden Connectortypen definiert sind (Quellcode A.5).

Quellcode A.5: *ModelTrait des Mehrphasenmodells*

```
template< int CollisionModel >
class MP3DTrait
{
public:
  typedef MP3DBlockGrid          BlockGridType;
  typedef MP3DBlockDescriptor    BlockDescriptorType;

  typedef MP3D_ScaleModule< CollisionModel > ScaleModule;

  typedef std::vector<double> SL_VecType;

  typedef MP3D_SL_DirectConn< SL_VecType  >             SL_DirectConnType;
  typedef MP3D_SL_VecConn< Transmitter< SL_VecType > > SL_VectorConnType;

  typedef CbVector<double>     ScaleVecType;

  typedef MP3D_CF_Conn< Transmitter< ScaleVecType >, ScaleModule > CF_VecConnType;
  typedef MP3D_FC_Conn< Transmitter< ScaleVecType >, ScaleModule > FC_VecConnType;
};
```

Der Algorithmus selbst verwendet intern bei der Connectorzuweisung ausschließlich abstrakte Typen (Quellcode A.6), die erst durch den C++-Compiler sukzessive mit dem im jeweiligen Trait angegebenen konkreten Typ ersetzt werden. Dadurch werden alle benötigten Klassen automatisch zur Übersetzungszeit generiert und zugleich hinsichtlich Typsicherheit überprüft. Diese Programmiertechnik reduziert zum einen den Programmieraufwand und bietet zum anderen eine vielseitige Verwendbarkeit.

Quellcode A.6: *Klasse D3Q19AssignLocalConnectorsVisitor*

```
template< typename ModelTrait >
class D3Q19AssignLocalConnectorsVisitor : IGridVisitor
{
public:
D3Q19AssignLocalConnectorsVisitor( int startLevel, int stopLevel )
     :  INodeVisitor( startLevel,stopLevel)
{
}

void setConnectors(AMR3DBlockGrid const& grid, AMR3DBlock const& block)
{
 ModelTrait::BlockGridType& grid = dynamic_cast<ModelTrait::BlockGridType&>( grid );
 /*...*/

 ModelTrait::BlockDescriptorType& bvd
         = dynamic_cast<ModelTrait::BlockDescriptorType&>( block.getDescriptor() );
  /*...*/

 //SameLevelDirectConnector
 SmartPtr<ModelTrait::SL_DirectConnType> c1(new ModelTrait::SL_DirectConnType(neighBvd));
 bvd.setConnector( c1 )
 /*...*/

 //SameLevelVectorConnector
 TransmitterPtr< ModelTrait::SL_VecType > > sender( new VectorSenderLocal/*...*/ );
 TransmitterPtr< ModelTrait::SL_VecType > > receiv( new VectorRecevierLocal/*...*/ );

 SmartPtr< Connector > c2( new ModelTrait::SL_VecConnType( sender,receiver ) );
 bvd.setConn( c2 );
 /*...*/

};
```

In der Anwendung wird die Klasse wie folgt verwendet:

Quellcode A.7: *Anwendung des D3Q19AssignLocalConnectorsVisitors*

```
1  //////////////////////////////////
2  // Anwendung
3  //////////////////////////////////
4  int main()
5  {
6      ////////////////////////////////////
7      // Mehrphase
8      ////////////////////////////////////
9      MP3DBlockGrid mp3dGrid;
10     /*...*/
11     D3Q19AssignLocalConnectorVisitor< MP3DTrait > mp3dAssignLocalConns;
12     mp3dGrid.adaptByVisitor( mp3dAssignLocalConns );
13     /*...*/
14     ////////////////////////////////////
15     // Einphase
16     ////////////////////////////////////
17     EP3DBlockGrid ep3dGrid;
18     /*...*/
19     D3Q19AssignLocalConnectorVisitor< EP3DTrait > ep3dAssignLocalConns;
20     ep3dGrid.adaptByVisitor( ep3dAssignLocalConns );
21     /*...*/
22 }
```

A5.3 RCF

A5.3.1 Serialisierbare Klassen

Anhand eines Minimalbeispiels soll die prinzipielle Vorgehensweise bei der Verwendung des RCF aufgezeigt werden (vgl. Abs. 13.2.5). Weitere Beispiele und Funktionalitäten sind unter [99] zu finden. Damit sich Klassen serialisieren und somit als Bytestrom über das Netzwerk versenden lassen, müssen diese die serialize-Methode implementieren. Mit dieser erfolgt sowohl die Konvertierung in das archivspezifische Format als auch die Rücktransformation aus diesem. Hierzu müssen alle zu übertragenen Attribute dem später versendeten Archiv übergeben werden (vgl. Quellcode A.8).

Quellcode A.8: *Serialisierbare Basisklasse Geo*

```
1  class Geo
2  {
3      int id;
4  public:
5      Geo(int id) : id(id) { }
6      virtual ~Geo() {}
7      int getID() { return id; }
8
9      template<class Archive>
10     void serialize(Archive& ar) {  ar & id; }
11 };
```

Bei diesen Attributen muss es sich nicht um Datenprimitive handeln. Insofern für die entsprechende Klasse eine Serialisierungsvorschrift existiert, ist sie kompatibel zu dem Archiv. Für Standardklassen wie STL-Container oder boost::shared_ptr [8] stellt das SF von Lindrud entsprechende Serialisierungsmethoden zur Verfügung. Bei abgeleiteten Klassen werden Basisklassen durch die Templatefunktion serializeParent berücksichtigt (Quellcode A.9).

Quellcode A.9: *Serialisierbare abgeleitete Klasse Sphere*

```
1  class Sphere : public Geo
2  {
3     int rad;
4     boost::shared_ptr<Point3D> midPoint;
5  public:
6     Sphere(int x1, int x2, int x3, int rad, int id)
7      : Geo(id), rad(rad), midPoint( new Point3D(x1,x2,x3) ) {}
8     template<class Archive>
9     void serialize(Archive& ar)
10    {
11       serializeParent< Geo >(ar, *this);
12       ar & rad;
13       ar & midPoint;
14       if( ar.isWriting() ) std::cout << time << " serialize" << std::endl;
15       if( ar.isReading() ) std::cout << time << " deserialize" << std::endl;
16    }
17 };
18
19 //Registrierung beim Serialisierungsframework SF (optional)
20 AUTO_RUN_NAMED( SF::registerType<Sphere>( "Sphere" )          , SF_Sphere    );
21 AUTO_RUN_NAMED( SF::registerBaseAndDerived< Geo, Sphere >()), SF_Sphere_BD );
```

Im Gegensatz zu den meisten anderen Middleware-Ansätzen müssen die Klassen für das Marshalling nicht von speziellen Interfaceklassen abgeleitet werden. Ein weiterer Vorteil ist, dass man mit der serialize-Methode und in Dateien schreibenden Archiven auf einfache Weise die Persistenz von Objekten realisieren kann.

A5.3.2 Client-Server

Die von einem RCF-Server angebotenen Methoden werden über RCF-Schnittstellen, die mittels spezieller Makros erstellt werden, zur Verfügung gestellt. In Quellcode A.10 ist für das Interface ISetSphere, das mit setSphere genau eine Methode besitzt, die entsprechende Definition in der RCF-Syntax dargestellt. Als Argument nimmt diese Methode ein Objekt vom Typ Sphere entgegen und gibt nach erfolgreicher Ausführung ein int-Wertobjekt zurück.

Quellcode A.10: *RCF-Interface ISetSphere*

```
1 RCF_BEGIN( ISetSphere, "ISetSphere" )
2  RCF_METHOD_R1( int, setSphere, Sphere )
3 RCF_END( ISetSphere )
```

Ein Server kann dabei eine Vielzahl von RCF-Schnittstellen verwalten, von denen jede bis zu 35 Methoden umfassen kann, die jeweils maximal acht Argumente entgegen nehmen können. Die vom Server publizierten Methoden werden im Allgemeinen nicht von diesem ausgeführt, sondern an Objekte weitergeleitet, die über die geforderte Funktionalität verfügen. Hierzu benötigt man eine konkrete Klasse, die die Schnittstelle implementiert. Eine Klasse, die das ISetSphere implementiert, ist in Quellcode A.11 zu finden. Diese nimmt eine Kugel entgegen, speichert diese in einer Liste und gibt ihre Identifikationsnummer zurück. Ein Beispiel für die Serverinitialisierung in einer Anwendung ist in Quellcode A.12 zu finden.

Quellcode A.11: *Konkrete Schnittstellenklasse ImplClass*

```
1  class ImplClass
2  {
3    std::list<Sphere> sphereList;
4  public:
5    int setSphere( Sphere s )
6    {
7       std::cout << time << " ImplClass: "<<s.getID() << std::endl;
8       sphereList.push_back( s );
9       return s.getID();
10   }
11 };
```

Quellcode A.12: *Serveranwendung*

```
1  int main()
2  {
3     RCF::RcfServer server( RCF::TcpEndpoint endp( "0.0.0.0"/*IP*/, 20000/*Port*/));
4     ImplClass impl;
5     server.bind< ISetSphere >(impl);
6     server.start();
7  }
```

Zuerst wird der allgemeine RcfServer erzeugt, der in dem Beispiel sämtliche über Port 20.000 eingehenden Anfragen bearbeitet. Um die setSphere-Methode zu veröffentlichen, wird eine Instanz der Schnittstellenklasse ImplClass erzeugt und über das Interface ISetSphere an den RcfServer gebunden. Nach dem Start des Servers werden die über diese Schnittstelle eintreffenden Anfragen an *impl* weitergeleitet und von dieser Objektinstanz bearbeitet.

Quellcode A.13: *Clientanwendung*

```
1  int main()
2  {
3     Sphere s(0,0,0,5,22);
4     RcfClient<ISetSphere> client( TcpEndpoint endp("127.0.0.1", 20000) );
5     std::cout << time << " client: " << client.setSphere(s);
6  }
```

Um sich mit dem Server zu verbinden, wird ein ISetSphere-unterstützender Client generiert (Quellcode A.13). Durch Übergabe der Verbindungsdaten verbindet sich dieser automatisch mit dem Server an der angegebenen Adresse. Sollte diese Verbindung nicht erstellt werden können, erhält der Anwender eine entprechende Fehlermeldung. Der Client sendet hier eine Kugel an den Server und gibt anschließend den von diesem zurückgesendeten Wert auf dem Bildschirm aus. Führt man das Beispiel aus, so erhält man folgende Bildschirmausgaben:

```
12:00.001 serialize


12:00.004 client: 22
```

(a) *Client*

```


12:00.002 deserialize
12:00.003 ImplClass: 22
```

(b) *Server*

Abbildung A.1: *Bildschirmausgaben von Server- und Clientanwendung*

Lebenslauf

Persönliche Daten

Name	Sören Freudiger
Geburtsdaten	22. Dezember 1974 in Berlin-Steglitz
Familienstand	Ledig, keine Kinder
Vater	Uwe Freudiger
Mutter	Ingrid Freudiger, geb. Dahms

Schulbildung

1981 - 1985	Grundschule Vagen
1985 - 1994	Gymnasium Bad Aibling Leistungskurse: • Mathematik • Wirtschafts- und Rechtslehre
1994 - 2001	Technische Universität München (Studium) Bauingenieurwesen (Dipl.-Ing.) Vertiefungsrichtungen: • Bauinformatik • Bau von Landverkehrswegen

Berufserfahrung

seit 2001	CAB/iRMB (TU Braunschweig) *Wissenschaftlicher Mitarbeiter* *(Softwareentwicklung im Bereich Strömungsmechanik)*
2000	USM (München) *Endkundenbetreuung*

1998 - 2000	Systhema (München) *Durchführung von Softwaretests* *Software- und Techniksupport* *Endkundenbetreuung*
1997	Intallbau/Meister (Dresden/Rosenheim) *Baupraktikum*
1996 -1997	Lehrstuhl für Bauinformatik TU München *Hilfswissenschaftler*
1996 -1998	Tychsen (Bruckmühl) *Durchführung von Siebdruck- und Montagearbeiten*
1995 -1996	Wrigleys (Rosenheim) *Vermarktung neuer Produkte und Kundenbetreuung*
1994-1995	Deimos (Westerham) *Reparatur von Computerbauteilen*
1989-1994	MTE (Valley) *Elektroinstallationen (u. a. für Flughafen München II)*

Veröffentlichungen

2008	*A parallelization concept for a multi-physics lattice Boltzmann solver based on hierarchical grids* Progress in Computational Fluid Dynamics
2007	VIRTUALFLUIDS*: Ein komponentenbasiertes Framework für parallele Lattice Boltzmann Simulationen auf hierarchischen Gittern* Forum Bauinformatik 2007 - Junge Wissenschaftler Forschen
2007	*Simulation von Strömungen mit freien Oberflächen auf blockstruktierten Gittern mit der Lattice Boltzmann Methode* Forum Bauinformatik 2007 - Junge Wissenschaftler Forschen
2006	*An adaptive scheme using hierarchical grids for lattice Boltzmann multi-phase flow simulations* Computers and Fluids
2004	*Microsoft .NET: Eine neue Entwicklungsplattform - auch für numerische Probleme?* Forum Bauinformatik 2004 - Junge Wissenschaftler forschen ISBN 3-8322-2022-4
2003	*Simulation von Mehrphasenströmungen mit der Lattice-Boltzmann-Methode auf hierarischen Gittern* Forum Bauinformatik 2003 - Junge Wissenschaftler forschen ISBN 3-8322-2022-4

2001	*Effiziente Datenstrukturen für Lattice-Boltzmann-Simulationen in der computergestützten Strömungsmechanik (C++)* Diplomarbeit
1998	*Effiziente Algorithmen zur Verbesserung von FEM-Netzen (C++)* Studienarbeit

Vorträge

2008	VIRTUALFLUIDS*: a component based framework for parallel, adaptive lattice Boltzmann simulations based on hierarchical block grids* ICMMES, Amsterdam, Niederlande
2007	VIRTUALFLUIDS*: Ein komponentenbasiertes Framework für parallele Lattice Boltzmann Simulationen auf hierarchischen Blockgittern* Forum Bauinformatik, Graz, Österreich
2007	VIRTUALFLUIDS*: A component based framework for parallel lattice Boltzmann simulations based on hierarchical block grids* ICMMES, Hampton, USA
2007	VIRTUALFLUIDS*: A parallel multi-physics lattice Boltzmann solver based on hierarchical block grids* parCFD, Antalya, Türkei
2006	*Parallelization of a Lattice Boltzmann prototype based on hierarchical grids* ICMMES, Hampton, USA
2006	*Lattice Boltzmann methods: from basics to applications* ASIM Workshop, München, Deutschland
2006	*Advanced LB methods (Workshop)* LSS, Erlangen, Deutschland
2005	*Efficient immiscible multiphase flow simulations hierarchical grids based on the Lattice Boltzmann method* Third M.I.T. Conference on Computational Fluid and Solid Mechanics, Boston, USA
2005	VIRTUALFLUIDS*: A node based LBM preprocessor* NEC, St. Augustin, Deutschland
2003	*Simulation von Mehrphasenströmungen mit der Lattice-Boltzmann-Methode auf hierarischen Gittern* Forum Bauinformatik 2003, Hannover, Deutschland
2003	*Lattice Boltzmann models for the simulation of multi-phase flows on hierachical grids* GAMM 2003, Venedig, Italien

Eidesstattliche Erklärung

Name:	Sören Freudiger
Geburtsdatum/-ort:	22. Dezember 1974 in Berlin-Steglitz
Straße:	Werderstr. 42
Ort:	69120 Heidelberg

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Dissertation mit dem Titel

Entwicklung eines parallelen, adaptiven, komponentenbasierten Strömungskerns für hierarchische Gitter auf Basis des Lattice-Boltzmann-Verfahrens

selbstständig verfasst, nicht schon als Diplom- oder Prüfungsarbeit verwendet und alle in Anspruch genommenen Hilfen in der Dissertation angegeben habe.

Ich versichere an Eides statt, dass ich diesen Promotionsantrag erstmalig einreiche und keine früheren Versuche einer Promotion unternommen habe.

_____

(Ort, Datum, Unterschrift)

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Index

Literaturverzeichnis

[1] 3-D-Modell: Capstone. Website, 2008. http://www-c.inria.fr/gamma/download, letzter Zugriff 22.10.2008.

[2] 3-D-Modell: TIE Defender. Website, 2009. http://mitglied.lycos.de/Mesh_Factor_Y/meshes/meshes.htm, letzter Zugriff 07.02.2009.

[3] Ahrenholz, B.: *Massively parallel simulations of multiphase- and multicomponent flows using lattice Boltzmann methods.* Dissertation, Institut für Rechnergestützte Modellierung im Bauingenieurwesen, Technische Universität Braunschweig, 2009.

[4] Akenine-Möller, T. und Haines, E.: *Real-Time Rendering Second Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2002.

[5] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, February 2001.

[6] Berg, I.: *muParser - a fast math parser library.* Website, 2005-2008. http://muparser.sourceforge.net, letzter Zugriff 07.02.2009.

[7] Bhatnagar, P. L., Gross, E. P. und Krook, M.: *A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems.* Physical Review, 94:511–525, 1954.

[8] boost - C++ libraries. Website. http://www.boost.org, letzter Zugriff 07.02.2009.

[9] Bornemann, F. A.: *An adaptive multilevel approach to parabolic equations III: 2D error estimation and multilevel preconditioning.* Impact of Computing in Science and Engineering., 4(1):1–45, 1992, ISSN 0899-8248.

[10] Bouzidi, M., Firdaouss, M. und Lallemand P.: *Momentum transfer of a lattice-Boltzmann fluid with boundaries.* Physics of Fluids, 13:3452–3459, 2001.

[11] Brüggemann, M.: *Informationsmodellierung im Bauwesen.* Habilitation, Lehrstuhl für Bauinformatik, Fakultät Architektur, Bauingenieurwesen und Stadtplanung, BTU Cottbus, 2007, ISBN 978-3-934934-12-2.

[12] Buick, J. und Greated, C. A.: *Gravity in a lattice Boltzmann model.* Physical Review E, 61:5307–5320, 2000.

[13] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. und Stal, M.: *Pattern: Broker.* Website, 2008. http://www.vico.org/pages/PatronsDisseny/PatternBroker, letzter Zugriff 07.02.2009.

[14] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P. und Stal, M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, August 1996, ISBN 0471958697.

[15] Cerami, E.: *Web Services Essentials.* O'Reilly, 2002.

[16] Cercignani, C., Illner, R. und Pulvirenti, M.: *The Mathematical Theory of Dilute Gases*, Bd. 106 d. Reihe *Applied Mathematical Sciences*. Springer Verlag, Berlin; New York, U.S.A., 1994.

[17] Cercignani, C. und Penrose, R.: *Ludwig Boltzmann: The Man Who Trusted Atoms*. Oxford University Press, 1998.

[18] CFD Online. Website, 2009. http://www.cfd-online.com, letzter Zugriff 07.02.2009.

[19] Clift, R., Grace, J. und Weber, M.: *Bubbles, Drops, and Particles*. Academic Press, San Diego, 1978.

[20] CodeProject. Website. http://www.codeproject.com, letzter Zugriff 07.02.2009.

[21] Constantinescu, G. und Squires, K.: *LES and DES investigations of turbulent flow over a sphere at Re=10000*. Flow, turbulence and combustion, 70(1–4):267–298, 2003.

[22] Crouse, B.: *Lattice-Boltzmann Strömungssimulationen auf Baumdatenstrukturen*. Dissertation, Lehrstuhl für Bauinformatik, Technische Universität München, 2003.

[23] Crouse, B., Rank, E., Krafczyk, M. und Tölke, J.: *A LB-Based Approach for Adaptive Flow Simulations*. International Journal of Modern Physics B, 17:109–112, 2003.

[24] Dave Winer: *XML-RPC Specification*. Website, 2009. http://www.xmlrpc.com/spec, letzter Zugriff 28.04.2009.

[25] Davis, D. und Parashar, M. P.: *Latency Performance of SOAP Implementations*. In: *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, S. 407, Washington, DC, USA, 2002. IEEE Computer Society, ISBN 0-7695-1582-7.

[26] debian: *Debian (GNU-Projekt)*. Website, 2009. http://www.debian.org, letzter Zugriff 30.03.2009.

[27] Dellar, P.: *Incompressible limits of lattice Boltzmann equations using multiple relaxation times*. Journal of Computational Physics, 190:351–370, 2003.

[28] Desplat, J. C., Pagonabarraga, I. und Bladon, P.: *LUDWIG: A parallel Lattice-Boltzmann code for complex fluids*. Computer Physics Communications, 134:273–290, März 2001.

[29] d'Humières, D., Ginzburg, I., Krafczyk, M., Lallemand, P. und Luo, L. S.: *Multiple-relaxation-time lattice Boltzmann models in three dimensions*. Royal Society of London Philosophical Transactions Series A, 360:437–451, 2002.

[30] Ding, H., Shu, C., Yeo, K. S. und Xu D.: *Numerical simulation of flows around two circular cylinders by mesh-free least square-based finite difference methods*. International Journal for Numerical Methods in Fluids, 53:305–332, 2007.

[31] Drepper, U.: *What Every Programmer Should Know About Memory*. Website, 2007. http://people.redhat.com/drepper/cpumemory.pdf, letzter Zugriff 07.02.2009.

[32] Düster, A., Bröker, H., Heidkamp, H., Heißerer, U., Kollmannsberger, S., Krause, R., Muthler, A., Niggl, A., Nübel, V., Rücker, M. und D., S.: *AdhoC^4 – User's Guide*. Lehrstuhl für Bauinformatik, Technische Universität München, 2004.

[33] Elfwing, R. und Paulsson, U.: *Performance of SOAP in Web Service Environment compared to CORBA*. Master Thesis, Blekinge Institute of Technology, 2002.

[34] Engelen, R. van: *gSOAP 2.7.11 User Guide*. Website, Aug. 2008. http://www.cs.fsu.edu/~engelen/soap.html, letzter Zugriff 07.02.2009.

[35] Enright, D., Fedkiw, R., Ferziger, J. und Mitchell, I.: *A hybrid particle level set method for improved interface capturing*. Journal of Computational Physics, 183(1):83–116, 2002, ISSN 0021-9991.

[36] Exa Corporation: *PowerFlow*. Website, 2009. http://www.exa.com, letzter Zugriff 27.03.2009.

[37] Filippova, O. und Hänel, D.: *Boundary-Fitting and Local Grid Refinement for Lattice-BGK Models*. International Journal of Modern Physics C, 9:1271–1279, 1998.

[38] Freeman, E., Freeman, E., Bates, B. und Sierra, K.: *Head First - Design Patterns*. O' Reilly & Associates, Inc., 2004, ISBN 0596007124.

[39] Freudiger, S.: *Effiziente Datenstrukturen für Lattice-Boltzmann-Simulationen in der computergestützten Strömungsmechanik*. Diplomarbeit, Lehrstuhl für Bauinformatik, Technische Universität München, 2001.

[40] Frisch, U., d'Humières, D., Hasslacher, B., Lallemand, P., Pomeau, Y. und Rivet, J. P.: *Lattice Gas Hydrodynamics in Two and Three Dimensions*. Complex Systems 1, S. 75–136, 1987.

[41] Fuchs, H. V.: *Schallabsorber und Schalldämpfer*. Springer Verlag, Berlin, 2007. ISBN:978-3-540-35493-2.

[42] Gabriel, E., Graham, E. F., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L. und Woodall, T. S.: *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*, S. 97–104, Budapest, Hungary, September 2004.

[43] Gamma, E., Helm, R., Johnson, R. und Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, Januar 1995.

[44] GCC: *Using the GNU Compiler Collection*. Website, 2009. http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc.pdf, letzter Zugriff 30.03.2009.

[45] GCC: *Using the GNU Compiler Collection - For gcc version 4.2.4*. Website, 2009. http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc.pdf, letzter Zugriff 21.03.2009.

[46] Geller, S.: *Fluid-Struktur-Interaktion mit der Lattice-Boltzmann Methode auf baumhierarchischen Gittern*. Dissertation, Institut für Rechnergestützte Modellierung im Bauingenieurwesen, Technische Universität Braunschweig, voraussichtliche Veröffentlichung 2009.

[47] Geller, S., Krafczyk, M., Tölke, J., Turek, S. und Hron, J.: *Benchmark computations based on Lattice-Boltzmann, Finite Element and Finite Volume Methods for laminar Flows*. Computers & Fluids, 35:888–897, 2006.

[48] Geller, S., Tölke, J. und Krafczyk, M.: *Lattice Boltzmann Methods on Quadtree-Type Grids for Fluid-Structure Interaction*. In: Bungartz, H. und Schäfer, M. (Hrsg.): *Fluid-Structure Interaction, Modelling, Simulation and Optimisation*, Bd. 53 d. Reihe *Lecture Notes in Computational Science and Engineering*, S. 270–293. Springer Verlag, Berlin, 2006, ISBN 978-3540345954.

[49] Ginzbourg, I. und Adler, P. M.: *Boundary flow condition analysis for the three-dimensional lattice Boltzmann model*. Journal de Physique II, 4:191–214, 1994.

[50] Ginzburg, I.: *Variably saturated flow described with the anisotropic Lattice Boltzmann methods*. Computers & fluids, 35(8-9):831–848, 2006.

[51] Ginzburg, I.: *Lattice Boltzmann modeling with discontinuous collision components: Hydrodynamic and Advection-Diffusion Equations.* Journal of Statistical Physics, 126(1):157–206, 2007, ISSN 0022-4715.

[52] Ginzburg, I. und d'Humières, D.: *Local second-order boundary methods for lattice Boltzmann models.* Journal of Statistical Physics, 84:927–971, 1996.

[53] Ginzburg, I. und d'Humières, D.: *Multi-reflection boundary conditions for lattice Boltzmann models.* Physical Review E, 68:066614.1–066614.30, 2003.

[54] Ginzburg, I., Verhaeghe, F. und d'Humières, D.: *Study of simple hydrodynamic solutions with the two-relaxation-times lattice Boltzmann scheme.* Communications in Computational Physics, 3:519–581, 2008.

[55] Ginzburg, I., Verhaeghe, F. und d'Humières, D.: *Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions.* Commun. Comput. Phys., 3:427–478, 2008.

[56] Ginzburg, I. und Wittum, G.: *Multigrid Methods for Two Phase Flows.* Numerical Flow Simulation (E. H. Hirschel ed.), Notes on Numerical Fluid Mechanics, 66:144–167, 1998.

[57] Goetz, J., Feichtinger, C., Iglberger, K., Donath, S. und Ruede, U.: *Large scale simulation of fluid structure interaction using Lattice Boltzmann methods and the 'physics engine.* In: Mercer, G. N. und Roberts, A. J. (Hrsg.): *Proceedings of the 14th Biennial Computational Techniques and Applications Conference, CTAC-2008*, Bd. 50 d. Reihe *ANZIAM J.*, S. C166–C188, Oktober 2008.

[58] Gray, N.: *Comparison of Web Services, Java-RMI, and CORBA service implementations.* Fifth Australian Workshop on Software and System Architectures, 2004.

[59] Gregor, D.: *Fixing Probe for Multi-Threaded MPI Applications.* Techn. Ber. 674, Indiana University Computer Science, Januar 2009.

[60] Grisby, D., Lo, S. L. und Riddoch, D.: *The omniORB version 4.1 User's Guide.* Website, 2007. http://omniorb.sourceforge.net, letzter Zugriff 07.02.2009.

[61] Gropp, W., Lusk, E., Ashton, D., Balaji, P., Buntinas, D., Butler, R., Chan, A., Goodell, D., Krishna, J., Mercier, G., Ross, R., Thakur, R. und Toonen, B.: *MPICH2 User's Guide - Version 1.0.8.* Website, 2008. http://www-unix.mcs.anl.gov/mpi/mpich, letzter Zugriff 07.02.2009.

[62] Gropp, W. und Thakur, R.: *Thread-safety in an MPI implementation: Requirements and analysis.* Parallel Computing, 33(9):595–604, 2007, ISSN 0167-8191.

[63] Grunau, D., Chen, S. und Eggert, K.: *A Lattice Boltzmann Model for multiphase flow.* Physics of Fluids, A5(10):2557–2561, 1993.

[64] Grundmann, T., Ritt, M. und Rosenstiel, W.: *TPO++: An object-oriented message-passing library in C++.* Website, 2007. http://tpo.sourceforge.net, letzter Zugriff 07.02.2009.

[65] Gueyffier, D., Li, J., Nadim, A., Scardovelli, R. und Zaleski, S.: *Volume-of-fluid interface tracking with smoothed surface stress methods for three-dimensional flows.* Journal of Computational Physics, 152(2):423–456, 1999, ISSN 0021-9991.

[66] Gunstensen, A. K. und Rothmann, D.: *Lattice Boltzmann model of immiscible fluids.* Physical Review A, 43(8):4320–4327, 1991.

[67] Guo, Z., Zheng, C. und Shi, B.: *Discrete lattice effects on the forcing term in the lattice Boltzmann method.* Physical Revue E, 65:46308, 2002.

[68] He, X. und Luo, L. S.: *Lattice Boltzmann model for the incompressible Navier-Stokes equation.* Journal of Statistical Physics, 88(3-4):927–944, 1997, ISSN 0022-4715.

[69] He, X. und Luo, L. S.: *Theory of the lattice Boltzmann method: from the Boltzmann equation to the lattice Boltzmann equation.* Physical Revue E, 56:6811–6817, 1997.

[70] He, X., Shan, X. und Doolen, G. D.: *Discrete Boltzmann equation model for nonideal gases.* Physical Review E, 57(1):R13–R16, Jan 1998.

[71] Hegewald, J., Krafczyk, M., Tölke, J., Hoekstra, A. G. und Chopard, B.: *An Agent-Based Coupling Platform for Complex Automata.* In: Bubak, M., Albada, G. D. van, Dongarra, J. und Sloot, P. M. A. (Hrsg.): *ICCS (2)*, Bd. 5102 d. Reihe *Lecture Notes in Computer Science*, S. 227–233. Springer Verlag, Berlin, 2008, ISBN 978-3-540-69386-4.

[72] Hegewald, Jan : *Project: MUSCLE.* Website, 2009. http://developer.berlios.de/projects/muscle, letzter Zugriff 25.02.2009.

[73] Henning, M.: *A New Approach to Object-Oriented Middleware.* IEEE Internet Computing, 8(1):66–75, 2004, ISSN 1089-7801.

[74] Henning, M.: *The rise and fall of CORBA.* Queue, 4(5):28–34, 2006, ISSN 1542-7730.

[75] Henning, M. und Spruiell, M.: *Distributed Programming with Ice.* Website, 2008. http://zeroc.com/download/Ice/3.3/Ice-3.3.0.pdf, letzter Zugriff 07.02.2009.

[76] Henning, M. und Vinoski, S.: *Advanced CORBA® Programming with C++.* Addison Wesley, 1999.

[77] Hirt, C. W. und Nichols, B. D.: *Volume of fluid method for the dynamics of free boundaries.* Journal of Computational Physics, 10:201–225, 1981.

[78] Hoekstra, A. G., Lorenz, E., Falcone, J. L. und Chopard, B.: *Towards a Complex Automata Framework for Multi-scale Modeling: Formalism and the Scale Separation Map.* In: *International Conference on Computational Science (1)*, Bd. 4487, S. 922–930. Springer Verlag, Berlin, 2007.

[79] Hou, S., Sterling, J., Chen, S. und Doolen, G. D.: *A Lattice Boltzmann Subgrid Model for High Reynolds Number Flows.* Contributions to Mineralogy and Petrology, S. 1004–+, 1994.

[80] Hyman, J.: *Numerical Methods for Tracking Interface.* Physica D, 12:396–407, 1984.

[81] Iglberger, K. und Ruede, U.: *Massively Parallel Rigid Body Dynamics Simulations.* Computer Science - Research and Development, zur Veröffentlichung akzeptiert, 2009.

[82] Intel Corporation: *Intel MPI Library.* Website, 2008. http://www.intel.com, letzter Zugriff 07.02.2008.

[83] Janßen, C. und Krafczyk, M.: *A Lattice Boltzmann approach for free surface flow simulations on non-uniform block-structured grids.* zur Veröffentlichung eingereicht 2009.

[84] Jenkins, L. N., Khorrami, M. R., Choudhari, M. M. und McGinley, C. B.: *Characterization Of Unsteady Flow Structures Around Tandem Cylinders For Component Interaction Studies In Airframe Noise.* 11th AIAA/CEAS Aeroacoustics Conference (26th Aeroacoustics Conference), Monterey, California, S. 1–16, 2005.

[85] Josuttis, N. M.: *SOA in Practice: The Art of Distributed System Design (Theory in Practice).* O'Reilly Media, Inc., August 2007, ISBN 0596529554.

[86] Junk, M. und Klar, A.: *Discretizations for the Incompressible Navier-Stokes Equations Based on the Lattice Boltzmann Method.* SIAM Journal on Scientific Computing, 22(1):1–19, 2000, ISSN 1064-8275.

[87] Junk, M., Klar, A. und Luo, L.: *Asymptotic analysis of the lattice Boltzmann equation.* Journal of Computational Physics, 210:676, 2005.

[88] Kandhai, D., Koponen, A., Hoekstra, A. G., Kataja, M., Timonen, J. und Sloot, P. M. A.: *Lattice-Boltzmann hydrodynamics on parallel systems.* Computer Physics Communications, 111:14–26, Juni 1998.

[89] Karypis, G. und Kumar, V.: *METIS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices - Version 4.0.* Website, 1998. http://glaros.dtc.umn.edu/gkhome/views/metis, letzter Zugriff 21.01.2009.

[90] Kehrwald, D.: *Numerical analysis of Immiscible Lattice BGK.* Dissertation, Universität Kaiserslautern, 2003.

[91] Kitware: *vtk - The Visualization Toolkit.* Website, 2008. http://www.vtk.org, letzter Zugriff 07.02.2009.

[92] Krafczyk, M.: *Gitter-Boltzmann-Methoden: Von der Theorie zur Anwendung.* Habilitation, Lehrstuhl für Bauinformatik, Fakultät für Bauingenieur- und Vermessungswesen, Technische Universität München, 2001.

[93] Krafczyk, M., Lehmann, P., Philippova, O., Hänel, D. und Lantermann, U.: *Lattice Boltzmann Simulations of complex Multi-Phase Flows.* In: *Multifield Problems: State of the art,* S. 50–57. Springer Verlag, Berlin, 2000.

[94] Krafczyk, M., Tölke, J. und Luo, L. S.: *Large-Eddy Simulations with a Multiple-Relaxation LBE Model.* International Journal of Modern Physics B, 17:33–39, 2003.

[95] Krafczyk, M. und Tölke, J.: *Lattice Boltzmann Methods - Basics and Recent Progress.* In: *Proceedings of NAFEMS workshop*, Niedernhausen, Germany, Mai 2003.

[96] Ladd, A. J. C.: *Numerical simulations of particulate suspensions via a discretized Boltzmann equation. Part 2. Numerical results.* Journal of Fluid Mechanics, 271:311–339, 1994.

[97] Lallemand, P. und Luo, L. S.: *Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability.* Physical Review E, 61(6):6546–6562, 2000.

[98] Lallemand, P. und Luo, L. S.: *Lattice Boltzmann method for moving boundaries.* Journal of Computational Physics, 184:406–421, 2003.

[99] Lindrud, J.: *RCF - Interprocess communication for C++.* Website, 2005-2008. http://deltavsoft.com/RcfUserGuide, letzter Zugriff 07.02.2009.

[100] Luo, L. S.: *Unified Theory of the lattice Boltzmann models for nonideal gases.* Physical Review Letters, 81(8):1618–1621, 1998.

[101] Lusk, E.: *Dynamic process management in an MPI setting.* In: *Proceedings of Seventh IEEE Symposium on Parallel and Distributed Processing*, S. 530. IEEE Computer Society Press, 1995.

[102] Marié, S., Ricot, D. und Sagaut, P.: *Comparison between lattice Boltzmann method and Navier Stokes high order schemes for computational aeroacoustics.* Journal of Computational Physics, 228:1056–1070, 2009.

[103] McCracken, M. E.: *Development and evaluation of lattice Boltzmann models for investigations of liquid break-up.* Dissertation, Purdue University, 2004.

[104] McNamara, G. und Zanetti, G.: *Use of the Boltzmann equation to simulate lattice-gas automata.* Physical Review Letters, 61:2332–2335, November 1988, ISSN 0031-9007.

[105] Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface*. Website, 2003. http://www.mpi-forum.org, letzter Zugriff 10.10.2008.

[106] Mittal, S. und Kumar, V.: *Vortex Induced Vibrations of a Pair of Cylinders at Reynolds Number 1000*. International Journal of Computational Fluid Dynamics, 18:601–614, Oktober 2004.

[107] Myricom: *MPICH-GM*. Website, 2009. http://www.myri.com/scs/download-mpichgm.html, letzter Zugriff 21.03.2009.

[108] Myricom: *Myricom - Pioneering Higher Performance Computing*. Website, 2009. http://www.myri.com, letzter Zugriff 18.03.2009.

[109] N. Ashgriz, J. P.: *FLAIR: Flux Line-Segment Model for Advection and Interface*. Journal of Computational Physics, 93:449–468, 1991.

[110] Nguyen, N. Q. und Ladd, A. J. C.: *Sedimentation of hard-sphere suspensions at low Reynolds number*. Journal of Fluid Mechanics, 525:73–104, Februar 2005.

[111] Nokia: *Qt Software*. Website, 2009. http://www.qtsoftware.com, letzter Zugriff 07.02.2009.

[112] Norfolk, M.: *Knottingley and Ferrybridge online*. Website, März 2009. http://www.knottingley.org, letzter Zugriff 01.03.2009.

[113] Ogayar, C. J., Segura, R. J. und Feito, F. R.: *Point in solid strategies*. Computers & Graphics, 29(4):616–624, August 2005.

[114] Oliker, L., Carter, J., Wehner, M., Canning, A., Ethier, S., Govindasamy, B., Mirin, A. und Parks, D.: *Leading Computational Methods on Scalar and Vector HEC Platforms*. In: *Proceedings SC2005*, Seattle, WA, 2005.

[115] OpenMP: *OpenMP Application Program Interface – Version 3.0*. Website, 2008. http://openmp.org/wp/openmp-specifications/spec30.pdf, letzter Zugriff 07.02.2009.

[116] Osher, S. und Fedkiw, R.: *Level set methods and dynamic implicit surfaces*. Springer Verlag, Berlin, 2003.

[117] Osher, S. und Sethian, J. A.: *Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations*. Journal of Computational Physics, 130:269–282, 1997.

[118] Pohl, T., Deserno, F., Thürey, N., Rüde, U., Lammers, P., Wellein, G. und Zeiser, T.: *Performance Evaluation of Parallel Large-Scale Lattice Bolzmann Applications on Three Supercomputing Architectures*. In: *Proceedings SC2004: High performance computing, networking, and storage conference*, Pittsburgh, PA, 6.-12. November 2004.

[119] Pohl, T., Kowarschik, M., Iglberger, K., Wilke, J. und Rüde, U.: *Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes*. Parallel Processing Letters, 13:549–560, 2003.

[120] Pope, S.: *Turbulent Flows*. Cambridge University Press, Cambridge, 2000. ISBN 0-444-88814-4.

[121] Puckett, E. G., Almgren, A. S., Bell, J. B., Marcus, D. L. und Rider, W. J.: *A high-order projection method for tracking fluid interfaces in variable density incompressible flows*. Journal of Computational Physics, 130(2):269–282, 1997, ISSN 0021-9991.

[122] Qian, Y. H., d'Humières, D. und Lallemand, P.: *Lattice BGK models for Navier-Stokes equation*. Europhysics Letters (EPL), 17(6):479–484, 1992.

[123] Raina, S.: *Emulation of a Virtual Shared Memory Architecture.* Dissertation, Department of Computer Science, University of Bristol, September 1993. http://www.cs.bris.ac.uk/Publications/Papers/1000020.pdf.

[124] Remedy IT: *TAO Programmers Guide.* Website, 2008. http://download.theaceorb.nl/TPG.pdf, letzter Zugriff 07.02.2009.

[125] Renardy, Y. und Renardy, M.: *PROST: A Parabolic Reconstruction of Surface Tension for the Volume of Fluid Method.* Journal of Computational Physics, 183:400–42, 2002.

[126] Rheinländer, M.: *A Consistent Grid Coupling Method for Lattice-Boltzmann Schemes.* Journal of Statistical Physics, 121(1-2):49–74, 2005.

[127] Rider, W. J. und Kothe, D. B.: *Reconstructing Volume Tracking.* Journal of Computational Physics, 141:112–152, 1998.

[128] Rothmann, D. H. und Keller, J. M.: *Immiscible Cellular Automaton Fluids.* Journal of Statistical Physics, 52(8):1119–1127, 1988.

[129] S. Popinet, S. Z.: *A front-tracking algorithm for accurate representation of surface tension.* International Journal for Numerical Methods in Fluids, 30:493–500, 1999.

[130] Sankaranarayanan, K. und Sundaresan, S.: *Lift force in bubbly suspensions.* Chemical engineering science, 57:3521–3542, 2002.

[131] Satofuka, N. und Nishioka, T.: *Parallelization of lattice Boltzmann method for incompressible flow computation.* Computational Mechanics, 23:164–171, 1999.

[132] Schelkle, M.: *LB-Verfahren zur Simulation dreidimensionaler Zweiphasen-Strömungen mit freien Oberflächen.* Dissertation, Universität Stuttgart, 1996.

[133] Schelkle, M., Rieber, M. und Frohn, A.: *Comparison of Lattice Boltzmann and Navier-Stokes simulations of three-dimensional free surface flows.* ASME Fluids Engineering Division Summer Meeting, Second International Symposium on Numerical Methods for Multiphase Flows, 1996.

[134] Schikuta, E. und Message-Passing-Interface-Forum: *MPI: A Message-Passing Interface Standard.* Techn. Ber., University of Tennessee, Knoxville, Tennesee, 1994.

[135] Schmidt, D., Stal, M., Rohnert, H. und Buschmann, F.: *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects.* John Wiley & Sons, September 2000, ISBN 0471606952.

[136] Schulz, M., Krafczyk, M., Tölke, J. und Rank, E.: *Parallelization Strategies and Efficiency of CFD Computations in complex geometries using Lattice-Boltzmann methods on High-Performance Computers.* In: Breuer, M., Durst, F. und Zenger, C. (Hrsg.): *Proceedings of the 3rd International FORTWIHR Conference on HPSEC,* S. 115–122. High-Performance Scientific and Engineering Computing, 12.-14. März 2002.

[137] Sethian, J.: *Level set methods: evolving interfaces in geometry, fluid mechanics, computer vision and materials science.* Cambridge University Press, 1996.

[138] Shan, X. und Chen, H.: *Lattice Boltzmann model for simulating flows with multiple phases and components.* Physical Review E, 47:1815–1819, 1993.

[139] Sharman, B., Lien, F. S., Davidson, L. und Norberg, C.: *Numerical predictions of low Reynolds number flows over two tandem circular cylinders.* SO: International Journal for Numerical Methods in Fluids, 47:423–447, 2005.

[140] Smagorinsky, J.: *General circulation experiments with the primitive equations: I. The basic experiment.* Monthly Weather Review, 91:99–164, 1963.

[141] Static Object Intersections. Website, 2009. http://www.realtimerendering.com/intersections.html, letzter Zugriff 07.02.2009.

[142] Stengel, H.: *C++-Programmiertechniken für High Performance Computing auf Systemen mit nichteinheitlichem Speicherzugriff unter Verwendung von OpenMP.* Diplomarbeit, Georg-Simon-Ohm Fachhochschule Nürnberg, 2007.

[143] Sterling, J. D. und Chen, S.: *Stability analysis of lattice Boltzmann methods.* Journal of Computational Physics, 123(1):196–206, 1996, ISSN 0021-9991.

[144] Stiebler, M., Tölke, J. und Krafczyk, M.: *Advection-diffusion lattice Boltzmann scheme for hierarchical grids.* Computers & Mathematics with Applications, 55(7):1576–1584, 2008, ISSN 0898-1221.

[145] Sumner, D., Richards, M. D. und Akosile, O. O.: *Two staggered circular cylinders of equal diameter in cross-flow.* Journal of Fluids and Structures, 20:255–276, Februar 2005.

[146] SUN: *Java.* Website, 2009. http://java.sun.com, letzter Zugriff 28.03.2009.

[147] Sussman, M., Almgren, A. S., Bell, J. B., Colella, P., Howell, L. H. und Welcome, M. L.: *An adaptive level set approach for incompressible two-phase flows.* Journal of Computational Physics, 148(1):81–124, 1999.

[148] Swift, M. R., Orlandini, E., Osborn, W. R. und Yeomans, J. M.: *Lattice Boltzmann simulation of liquid-gas and binary fluid systems.* Physical Review E, 54:5041–5052, 1996.

[149] Szalmás, L.: *Slip on curved boundaries in the Lattice Boltzmann Model.* International Journal of Modern Physics C, 18(1):15–24, 2007.

[150] The Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1.* Website, 2008. http://www.omg.org/spec/CORBA/3.1, letzter Zugriff 07.02.2009.

[151] Thömmes, G., Becker, J., Junk, M., Vaikuntam, A. K., Kehrwald, D., Klar, A., Steiner, K. und Wiegmann, A.: *A lattice Boltzmann method for immiscible multiphase flow simulations using the level set method.* Journal of Computational Physics, 228(4):1139–1156, 2009, ISSN 0021-9991.

[152] Thömmes, P.: *Notizen zu C++.* Springer Verlag, Berlin, 2003.

[153] Thürey, N.: *A single-phase free-surface Lattice-Boltzmann Method.* Diplomarbeit, Universität Erlangen-Nürnberg, 2003.

[154] Tölke, J.: *Gitter-Boltzmann-Verfahren zur Simulation von Zweiphasenströmungen.* Dissertation, Lehrstuhl für Bauinformatik, Technische Universität München, München, 2001.

[155] Tölke, J. und Krafczyk, M.: *Towards three-dimensional Teraflop-CFD computations on a desktop PC.* Progress in Computational Fluid Dynamics, S. 46308, akzeptiert zur Veröff. 2008.

[156] Tölke, J.: *A thermal model based on the lattice Boltzmann method for low Mach number compressible flows.* Journal of Computational and Theoretical Nanoscience, 3(4):579–587, 2006.

[157] Tölke, J.: *Numerische 3D Modelle - Lattice-Boltzmann Methode.* Berichte des Lehrstuhls und der Versuchsanstalt für Wasserbau und Wasserwirtschaft, 104, 2006.

[158] Truckenbrodt, E.: *Fluidmechanik Band 1: Grundlagen und elementare Strömungsvorgänge dichtebeständiger Fluide.* Springer-Verlag, Berlin, 1996.

[159] Turek, S. und Schäfer, M.: *Benchmark computations of laminar flow around cylinder*. In: Hirschel, E. (Hrsg.): *Flow Simulation with High–Performance Computers II*, Bd. 52 d. Reihe *Notes on Numerical Fluid Mechanics*, S. 547–566. Vieweg Verlag, 1996. co. F. Durst, E. Krause, R. Rannacher.

[160] ubuntu: *Linux*. Website, 2009. http://www.ubuntu.com, letzter Zugriff 21.03.2009.

[161] Vandevoorde, D. und Josuttis, N. M.: *C++ Templates: The Complete Guide*. Addison-Wesley Professional, November 2002.

[162] W3C: *SOAP Version 1.2*. Website, 2009. http://www.w3.org/TR/soap, letzter Zugriff 28.04.2009.

[163] Williams, S., Carter, J., Oliker, L., Shalf, J. und Yelick, K.: *Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms*. IEEE International Parallel and Distributed Processing Symposium, 2008.

[164] Willms, A.: *C++ - verstehen, anwenden, erweitern*. Galileo Press, Bonn, 2000.

[165] Wolf-Gladrow, D.: *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer Verlag, Berlin, Heidelberg, 2000.

[166] Yu, D., Mei, R., Luo, L. S. und Shyy, W.: *Viscous flow computations with the method of lattice Boltzmann equation*. Progress in Aerospace Sciences, 39:329–367, 2003.

[167] Yu, H., Luo, L. S. und Girimaji, S. S.: *LES of turbulent square jet flow using an MRT lattice Boltzmann model*. Computers & Fluids, 35(8-9):957–965, 2006. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.